ELSEVIER

Theoretical Computer Science 155 (1996) 179–219

Theoretical Computer Science

From CML to its process algebra

Flemming Nielson *, Hanne Riis Nielson

Computer Science Department, Aarhus University, Ny Munkegade, DK-8000 Aarhus C, Denmark

Received October 1993; revised August 1994
Communicated by U. Montanari

Abstract

Reppy's language CML extends Standard ML of Milner et al. with primitives for communication. It thus inherits a notion of strong polymorphic typing and may be equipped with a structural operational semantics. As a first step we formulate an effect system for statically expressing the communication behaviours of CML programs as these are not reflected in the types. As a second step we adapt the structural operational semantics of CML so as to incorporate behaviours. We then show how types and behaviours evolve in the course of computation: types may decrease and behaviours may lose prefixes as well as decrease. As the syntax of behaviours is rather similar to that of a process algebra our main result may therefore be viewed as regarding the semantics of a process algebra as an *abstraction* of the semantics of an underlying programming language. This establishes a new kind of connection between "realistic" concurrent programming languages and "theoretical" process algebras.

1. Introduction

One trend in the research on process algebras is to extend them with "higher-order" features somewhat analogous to the "higher-order" role that functions play in functional languages. Some approaches allow passing labels or ports, e.g. [15], whereas others allow passing processes, e.g. [29, 31]. Sometimes this leads to hybrid calculi that contain the syntax of a process algebra as well as that of the λ-calculus, e.g. [7, 17]. Putting more emphasis on the functional features, another approach is to extend a "realistic" functional language with primitives for communication. Good examples include CML [15, 24–26], Facile [10], and LCS [2], but also Concurrent Clean [22] may be viewed in this way. We refer to [12] for a much more detailed survey of some of these issues.

We follow the latter approach and base ourselves on Reppy's language CML. It is an extension of Standard ML with primitives for communication; among other things

* Corresponding author. Email: {fnielson, hrnielson}@daimi.aau.dk.

SSDI 0304-3975(95)00017-8

this allows channels to be created and processes to be forked and then processes may send and receive values over channels. Since CML is an extension of Standard ML it inherits a notion of strong typing; this already distinguishes CML from several other approaches that are untyped in nature. However, the types are very close to those of Standard ML and therefore do not contain much information about the communication that takes place during computation. We believe it is desirable to "extend the notion of types" so as to give a concise summary of the possible communication behaviours. This is in line with the ideas of [17] but we deviate from [17] in separating the type and communication information by using the notion of effect system previously developed for functional languages, e.g. [13, 28]. Section 2 gives a presentation of this system.

Both [25, 26] and [1] give a structural operational semantics for CML. As is usual the types do not influence the semantics but for the purpose of proofs it may be desirable to label the transition relation with additional book-keeping details (and to retain some type information in the expressions). The main difference between [25, 26] and [1] is that the latter is a traditional operational semantics whereas the former uses the notion of "evaluation context" [9] in order to present the rules more concisely and in order to facilitate proofs. In Section 3 we present a definition close to that of [25, 26] but with additional book-keeping details; in keeping with tradition the types and behaviours do not influence the semantics.

The impact of the operational semantics on types and behaviours emerges when showing "subject reduction" and related results. Actually, types may *decrease* in the course of computation and this phenomenon also arose in [5] in the context of modelling object-oriented programming. In a similar way the behaviours may decrease in the course of computation but additionally certain prefixes may *disappear* due to the communications taking place. It is instructive to regard this combined decreasing and disappearance of behaviours as an *operational semantics* for behaviours. Since the behaviours syntactically resemble a process algebra (e.g. the one in [11]) this suggests the viewpoint that the semantics of a process algebra is an *abstraction* of the semantics of an underlying programming language. This is quite unlike previous attempts to formally relate programming languages to process algebras (as opposed to mixing their syntax as in [2, 7, 10, 15, 17, 26]) where programs are directly *translated* into terms of some given process algebra; this view is explicit in [14] and would seem to be implicit also in [11] (that presents the process algebra closest to our notion of behaviours). Section 4 provides the precise formulations of the results we have to offer as well as overviews of the proof techniques. It concludes by a discussion of the advantages and disadvantages of the "structural equivalence" approach to semantics of process algebras.

We finish with prospects for future research and concluding remarks in Section 5. In Appendix A we briefly discuss variations on the system presented here and in Appendix B we provide full details of the proofs. This paper subsumes the extended abstract that appeared as [19].

2. CML with behaviours of communication

We follow [1, 25, 26] in embedding the essential features of CML into a small fragment of Standard ML. For simplicity we restrict the attention to a monomorphic fragment and we take care to structure the syntax in a way that facilitates adding new constructs as the need arises.

The syntax of expressions $e \in$ **Exp** and weakly[1] evaluated expressions $w \in$ **WExp** is given by:

$$
\begin{aligned}
e &::= w \mid e_1\, e_2 \mid \texttt{let}\ i = e_1\ \texttt{in}\ e_2 \mid \texttt{rec}\ i_0(i_1) : t \Rightarrow e \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t \\
w &::= c : t \mid i \mid \texttt{fn}\ i : t \Rightarrow e \mid \cdots
\end{aligned}
$$

They are defined by mutual recursion and include constants (with an explicit monotype), identifiers[2] $i \in$ **Ident** (unspecified), function abstraction, application, `let`-abstraction but without any polymorphism, recursive definitions (with an explicit monotype indicating the type of the recursive function), and conditional (with an explicit monotype indicating the type of the result). The need for explicit monotypes will be clarified later in this section; here it suffices to say that our approach is consistent with [5] and that the development of [20, 21] allows to incorporate polymorphism by settling for "coarser" information. The three dots in the syntax serve as a reminder for the need to introduce (in the next section) additional weakly evaluated expressions corresponding to the intermediate results that arise during computation.

The syntax of constants $c \in$ **Const** is given by:

$$
\begin{aligned}
c ::= {} & ()\mid \texttt{true} \mid \texttt{false} \mid n \\
\mid {} & \texttt{pair} \mid \texttt{fst} \mid \texttt{snd} \\
\mid {} & \texttt{nil} \mid \texttt{cons} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{isnil} \\
\mid {} & \texttt{send} \mid \texttt{receive} \mid \texttt{choose} \mid \texttt{noevent} \\
\mid {} & \texttt{sync} \mid \texttt{wrap} \mid \texttt{fork} \mid \texttt{channel}
\end{aligned}
$$

This includes the element `()` of the unit type, the booleans `true` and `false`, and numerals $n \in$ **Num** (unspecified). For products we write `pair` $e_1\ e_2$ for (e_1, e_2) and we then use `fst` and `snd` to select components. Similarly for lists we write `cons` e_1 $(\cdots(\texttt{cons}\ e_n\ \texttt{nil})\cdots)$ for $[e_1, \ldots, e_n]$ and we select components using `hd` and `tl` and test for emptiness using `isnil`. To obtain a more readable notation we shall allow one to use (e_1, e_2) and $[e_1, \ldots, e_n]$ in examples.

Turning to the concurrency primitives we may send a value `v` over a channel `ch` by `sync(send(ch,v))`, receive a value over a channel `ch` by `sync(receive(ch))`, and choose between a list $[e_1, \ldots, e_n]$ of communications by `sync(choose(`$[e_1, \ldots, e_n]$`))` where the case $n = 0$ is written `sync(noevent)` and acts as a blocking statement.

[1] This terminology is consistent with the weak normal forms of [23].

[2] It is customary to take $w ::= i$ rather than $e ::= i$ but for the purposes of this paper the choice does not matter.

Here the primitives send, receive, choose and noevent do not actually perform the communications but produce *delayed communications* (much like *closures* or *thunks*) that are then *activated* by the sync operator (much like closures or thunks may be evaluated by applying them to a dummy argument). The rationale behind this choice of primitives is discussed in [25] and is beyond the scope of this paper. The operation $\texttt{wrap}(e_1, e_2)$ then modifies the delayed communication e_1 to another that applies e_2 to the resulting value; so $\texttt{sync}(\texttt{wrap}(e_1, e_2))$ may be thought of as $e_2(\texttt{sync}(e_1))$ provided that e_2 performs no communications. Finally, we may fork a process to the pool of processes and we may allocate a new free channel to be used for communication.

For types $t \in \mathbf{Type}$ we take:

$$t ::= \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid tv \mid t_1 \times t_2 \mid t\ \texttt{list} \mid t_1 \rightarrow^b t_2 \mid t\ \texttt{chan}\ r \mid t\ \texttt{com}\ b$$

As in Standard ML we have three base types, type variables $tv \in \mathbf{TyVar}$ (e.g. τ, τ', τ_1) and products and list. Concerning functions we use a superscript behaviour $b \in \mathbf{Beh}$ for indicating the communication that will take place when the function is evaluated; the precise details follow shortly. Much as in CML we have a type for channels over which values of a given type may be communicated; to allow some separation among the identity of channels we indicate the specific region where the channel is allocated. For regions $r \in \mathbf{Reg}$ we take:

$$r ::= i \mid r_1 + r_2 \mid rv$$

A region will describe a non-empty set of "program points" and we shall occasionally need region variables $rv \in \mathbf{RegVar}$ (e.g. ρ, ρ', ρ_1). However, it would be possible to dispense with regions throughout without invalidating the results of the paper. Also as in CML we have a type for a "delayed" communication yielding a result of a certain type; unlike CML we have added a behaviour for indicating the communication that will take place when the "delayed" communication is enacted.

Finally, behaviours $b \in \mathbf{Beh}$ are given by:

$$b ::= \varepsilon \mid r!t \mid r?t \mid t\ \textsc{chan}\ r \mid t\ \textsc{fork}\ b \mid b_1; b_2 \mid b_1 + b_2 \mid \textsc{rec}\ bv.b \mid bv$$

The behaviours include primitive constructs for describing "no communication", sending a value of some type over a channel allocated in a certain region, receiving a value, allocating a channel, and forking a new process of a given type and with a given behaviour when evaluated. We use semicolon to express that one behaviour takes place before another and we use plus to express that either the first behaviour takes place or the second does. For recursive functions we need a behaviour $\textsc{rec}\ bv.b$ for expressing a behaviour that is as given by b provided that recursive calls are as given by $bv \in \mathbf{BehVar}$ (e.g. β, β', β_1).

Example 2.1. The map function for mapping a function down a list of elements may be defined by

```
rec map f ⇒ fn xs ⇒ if isnil xs then nil
                    else cons (f (hd xs)) (map f (tl xs))
```

where we have dispensed with the explicit monotypes (i.e. the ": t") at a number of places. The overall type of map is

$$(\texttt{int} \to^{\varepsilon} \texttt{bool}) \to^{\varepsilon} \texttt{int list} \to^{\varepsilon} \texttt{bool list}$$

where we regard list (and product) as binding more tightly than $\to$. A parallel version may be defined by

```
rec mappar f ⇒ fn xs ⇒ if isnil xs then nil
                       else let ch = channel ()
                            in fork (fn d ⇒ sync
                                        (send (ch, f (hd xs))));
                               let ys = mappar f (tl xs)
                               in sync (wrap (receive ch,
                                              fn y ⇒ cons y ys))
```

where we write (e_1, e_2) for pair e_1 e_2 and $e_1; e_2$ for snd(e_1, e_2). Here a new channel is allocated and a new process is forked for processing each function in the list xs; when receiving the value for some function we use wrap to prepend it to the results produced by the subsequent functions. The overall type of mappar is

$$(\texttt{int} \to^{\varepsilon} \texttt{bool}) \to^{\varepsilon} \texttt{int list} \to^{b} \texttt{bool list}$$

where $b = \text{REC } \beta.\varepsilon + ((\texttt{bool CHAN m}); (\texttt{bool FORK m!bool}); \beta ; (\texttt{m?bool}))$ and where we assume that the region corresponding to the channel is m.

2.1. *Well-typing*

We shall say that an expression e has type t and behaviour b, written

$$tenv \vdash e \mid t \,\&\, b,$$

whenever the type of e is t in the usual sense and evaluation of e gives rise to the communication behaviour b. As usual *tenv* is a type environment, i.e. a finite list of pairs of identifiers and types, giving the types of free variables; since CML is an eager language there is no effect associated with accessing an identifier and therefore the type environment does not contain any behaviour component (except embedded within the types).

For constants our syntax prescribes an explicit monotype to be given; we use the polytypes of Fig. 1 to restrict the choice of monotypes. For the primitives also to be

found in Standard ML the polymorphic type listed is as usual except than an ε is placed on all function arrows to reflect that no communication is taking place. For the primitives of CML only three involve function arrows with a non-trivial behaviour (i.e. distinct from ε): `sync` that extracts the delayed communication of the argument and enacts it, `fork` that forks a new process and `channel` that allocates a new channel. The remaining primitives only construct or modify delayed communications without actually performing any communication; this is reflected by an ε on all function arrows. Taking `wrap` as an example its type clearly indicates that the delayed communication $\texttt{wrap}(e_1, e_2)$ first performs the communications of e_1, then the internal communications of e_2, while modifying the result of e_1 by the function e_2.

The details of the type inference for expressions are given by the axioms and rules of Fig. 2. We already explained the axioms for identifiers and constants. For function abstraction the resulting type and behaviour indicate that no communication takes place when constructing the function abstraction but only when the function is evaluated. For application the overall behaviour expresses eager left-to-right evaluation: first the expression in function position in evaluated to a function abstraction, then the argument is evaluated and finally the function is applied to the argument. We do not require equality between the type of the actual parameter and the type of the formal parameter

c	TypeOf(c)
`()`	`unit`
`true`	`bool`
`false`	`bool`
n	`int`
`pair`	$\forall\tau_1,\tau_2.\ \tau_1 \to^{\varepsilon} \tau_2 \to^{\varepsilon} \tau_1 \times \tau_2$
`fst`	$\forall\tau_1,\tau_2.\ \tau_1 \times \tau_2 \to^{\varepsilon} \tau_1$
`snd`	$\forall\tau_1,\tau_2.\ \tau_1 \times \tau_2 \to^{\varepsilon} \tau_2$
`nil`	$\forall\tau.\ \tau$ `list`
`cons`	$\forall\tau.\ \tau \to^{\varepsilon} \tau$ `list` $\to^{\varepsilon} \tau$ `list`
`hd`	$\forall\tau.\ \tau$ `list` $\to^{\varepsilon} \tau$
`tl`	$\forall\tau.\ \tau$ `list` $\to^{\varepsilon} \tau$ `list`
`isnil`	$\forall\tau.\ \tau$ `list` $\to^{\varepsilon}$ `bool`
`send`	$\forall\rho,\tau.\ (\tau$ `chan` $\rho) \times \tau \to^{\varepsilon} (\tau$ `com` $(\rho!\tau))$
`receive`	$\forall\rho,\tau.\ (\tau$ `chan` $\rho) \to^{\varepsilon} (\tau$ `com` $(\rho?\tau))$
`choose`	$\forall\beta,\tau.\ (\tau$ `com` $\beta)$ `list` $\to^{\varepsilon} (\tau$ `com` $\beta)$
`noevent`	$\forall\beta,\tau.\ (\tau$ `com` $\beta)$
`wrap`	$\forall\beta_1,\beta_2,\tau_1,\tau_2.\ (\tau_1$ `com` $\beta_1) \times (\tau_1 \to^{\beta_2} \tau_2) \to^{\varepsilon} (\tau_2$ `com` $(\beta_1;\beta_2))$
`sync`	$\forall\beta,\tau.\ (\tau$ `com` $\beta) \to^{\beta} \tau$
`fork`	$\forall\beta,\tau.\ (\texttt{unit} \to^{\beta} \tau) \to^{\tau\ \text{FORK}\ \beta}$ `unit`
`channel`	$\forall\iota,\tau.$ `unit` $\to^{\tau\ \text{CHAN}\ \iota} (\tau$ `chan` $\iota)$

Fig. 1. Types of primitives.

but merely that the type of the actual parameter is a subtype of the type of the formal parameter. The notion of subtype is developed below and as is illustrated in Appendix A this is useful for allowing a function expressing mild restrictions on the argument, e.g. that it only communicates over channels in certain regions, to be applied to a concrete argument with a very specific communication behaviour.

The rule for `let`-abstraction is rather straightforward due to the absence of polymorphism. The rule for recursive functions is much as the rule for function abstraction except that we need to extend the type environment with assumptions about the recursive function and we only require the type and behaviour of the body to be subtypes and subbehaviours of the corresponding parts of the assumptions. Example 2.1 above illustrates that REC-behaviours may be "deeply" nested within the type[3] of the recursive function.

Finally, the rule for conditional allows the types of the branches to be dissimilar and only requires them to be subtypes of a common and explicitly given monotype. To require equality would invalidate the subject reduction property proved in Section 4. To dispense with the explicitly given monotype would require a join-semilattice structure on types due to the contravariance of function space; this would require the behaviours to enjoy not only the join-semilattice structure developed below, but also a meet-semilattice structure and we have refrained from these complications.

Since we have integrated "subsumption" into the rules where needed, rather than having a general subsumption rule, the presence of explicit monotypes guarantees that types and behaviours are unique:

Fact 2.2 (Unique typing). *If* $tenv \vdash e \mid t_1 \,\&\, b_1$ *and* $tenv \vdash e \mid t_2 \,\&\, b_2$ *then* $t_1 = t_2$ *and* $b_1 = b_2$.

Proof. The proof is by induction on the inference of $tenv \vdash e \mid t_1 \,\&\, b_1$. □

Remark. Intuitively the sequential subset of CML only needs behaviours generated from ε using sequencing, sum and recursion. If we were to add the "rearrangement" rule

$$\frac{tenv \vdash e \mid t_1 \,\&\, b_1}{tenv \vdash e \mid t_2 \,\&\, b_2} \quad \text{if } t_1 \equiv t_2 \text{ and } b_1 \equiv b_2$$

(where $\equiv$ are the subtype and subbehaviour equivalences) we could simplify all such behaviours to ε. Also we would continue to have a form of unique types but only "modulo $\equiv$": the '=' of Fact 2.2 would have to be replaced by '$\equiv$'.

[3] In the notation of the `rec`-rule of Fig. 2 there need not be any occurrences of REC in b even though there may be occurrences in t.

$$tenv \vdash c : t \mid t\&\varepsilon \qquad \text{if } \mathrm{TypeOf}(c) \succ t$$

$$tenv \vdash i \mid t\&\varepsilon \qquad \text{if } tenv(i) = t$$

$$\frac{tenv[i \mapsto t] \vdash e \mid t'\&b}{tenv \vdash \mathtt{fn}\ i : t \ \Rightarrow\ e \mid t \to^b t'\&\varepsilon}$$

$$\frac{tenv \vdash e_1 \mid t \to^b t'\&b_1 \quad tenv \vdash e_2 \mid t^-\&b_2}{tenv \vdash e_1\ e_2 \mid t'\&b_1;b_2;b} \qquad \text{if } t^- \leqslant t$$

$$\frac{tenv \vdash e_1 \mid t_1\&b_1 \quad tenv[i \mapsto t_1] \vdash e_2 \mid t_2\ \&\ b_2}{tenv \vdash \mathtt{let}\ i{=}e_1\ \mathtt{in}\ e_2 \mid t_2\ \&\ b_1;b_2}$$

$$\frac{tenv[i_0 \mapsto t_1 \to^b t][i_1 \mapsto t_1] \vdash e \mid t^-\&b^-}{tenv \vdash \mathtt{rec}\ i_0(i_1) : t_1 \to^b t \Rightarrow e \mid t_1 \to^b t\&\varepsilon} \qquad \text{if } t^- \leqslant t \text{ and } b^- \leqslant b$$

$$\frac{tenv \vdash e \mid \mathtt{bool}\&b \quad tenv \vdash e_1 \mid t_1\&b_1 \quad tenv \vdash e_2 \mid t_2\&b_2}{tenv \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : t \mid t\&b;(b_1 + b_2)} \qquad \text{if } t_1 \leqslant t \text{ and } t_2 \leqslant t$$

Fig. 2. Type Inference.

2.2. Subtyping

Since types involve regions as well as behaviours the subtyping relation must involve a subregion relation and a subbehaviour relation. These relations may be defined by axioms and inference rules and have some important similarities (as well as important differences). To save repetition and to help demonstrating that they constitute the "right" collection we shall organize their presentation with diligence.

We begin with regions. Intuitively, $r_1 \leqslant r_2$ is to mean that the set of identifiers listed in r_1 is a subset of those listed in r_2. Formally, this may be axiomatized as shown in Fig. 3. The first 5 axioms and rules simply state that $\leqslant$ is a preorder and that $\equiv$ is the associated equivalence. The last 4 axioms and rules state that $+$ is a least upper bound operator (modulo the equivalence). The two axioms involving $\leqslant$ are standard but the inference rule and the axiom involving $\equiv$ are usually replaced by a rule that allows one to infer $r_1 + r_2 \leqslant r$ from $r_1 \leqslant r$ and $r_2 \leqslant r$. Luckily, the two formulations are equivalent in the presence of the other rules and axioms but we prefer the formulation chosen since the structural rule is typical of the rules we shall need for behaviours and types. The notion of polarity is explained below.

Turning to types we once more need to state that $\leqslant$ is a preorder and $\equiv$ is the associated equivalence. The details of this are as for regions and are therefore not repeated in Fig. 4. Next comes a structural rule for each type constructor. To summarize these succinctly we use the notion of polarity. There are three polarities: $\oplus$ for a covariant or monotonic position, $\ominus$ for a contravariant or antimonotonic position and $\odot$ for a mixed co- and contravariant position. The examples given in Fig. 3 and 4

$\leqslant$ is a preorder

$$r \leqslant r \qquad \frac{r_1 \leqslant r_2 \quad r_2 \leqslant r_3}{r_1 \leqslant r_3}$$

$\equiv$ is the associated equivalence

$$\frac{r_1 \leqslant r_2 \quad r_2 \leqslant r_1}{r_1 \equiv r_2} \qquad \frac{r_1 \equiv r_2}{r_1 \leqslant r_2} \qquad \frac{r_1 \equiv r_2}{r_2 \leqslant r_1}$$

structural rule with polarity $\oplus + \oplus$

$$\frac{r_1 \leqslant r_1^+ \quad r_2 \leqslant r_2^+}{r_1 + r_2 \leqslant r_1^+ + r_2^+}$$

$+$ is join

$$r_1 \leqslant r_1 + r_2 \qquad r_2 \leqslant r_1 + r_2 \qquad r \equiv r + r$$

Fig. 3. Coercion rules for regions.

$\leqslant$ is a preorder

$\equiv$ is the associated equivalence

structural rules with polarities

$$\oplus \times \oplus,\ \oplus\ \texttt{list},\ \ominus \rightarrow^{\oplus} \oplus,\ \odot\ \texttt{chan}\ \oplus,\ \oplus\ \texttt{com}\ \oplus$$

e.g.
$$\frac{t_1^- \leqslant t_1 \quad b \leqslant b^+ \quad t_2 \leqslant t_2^+}{t_1 \rightarrow^b t_2 \leqslant t_1^- \rightarrow^{b^+} t_2^+} \qquad \frac{t \equiv t' \quad r \leqslant r^+}{t\ \texttt{chan}\ r \leqslant t'\ \texttt{chan}\ r^+}$$

Fig. 4. Coercion rules for types.

should make the intention clear.[4] The definitions are in good accord with the literature on subtyping.

Many of the rules and axioms for behaviours in Fig. 5 follow the pattern seen already. The polarity laws for '+' and ';' are crucial for our approach; the remaining polarity laws are included to mimick the polarity laws for types but are not essential for the theoretical development. That '+' is join and that ';' and 'ε' constitute a monoid is crucial for the development; note that the axiom $\varepsilon; b \equiv b$ is reminiscent of the observational equivalence $\tau.b \approx b$ in CCS but we do not wish to claim that ε and τ model the same phenomenon. The rules and axioms for recursion are useful for examples – and further laws are studied in Appendix A – but they do *not* influence

[4] One can be more formal as follows. Write $t[\oplus]t'$ for $t \leqslant t'$, $t[\ominus]t'$ for $t' \leqslant t$, and $t[\odot]t'$ for $t \equiv t'$. A type constructor φ has polarity $\varphi(p_1, \ldots, p_n)$ if and only if the structural inference rule says that $\varphi(t_1, \ldots, t_n)[\oplus]\varphi(t_1', \ldots, t_n')$ follows from $t_1[p_1]t_1', \ldots, t_n[p_n]t_n'$.

$\leqslant$ is a preorder

$\equiv$ is the associated equivalence

structural rules with polarities

$$\oplus!\oplus,\ \oplus?\ominus,\ \odot\ \text{CHAN}\oplus,\ \oplus\ \text{FORK}\oplus,\ \oplus;\oplus,\ \oplus+\oplus$$

$+$ is join

; and ε constitute a monoid (modulo the equivalence)

$$b_1;(b_2;b_3)\equiv(b_1;b_2);b_3 \qquad b;\varepsilon\equiv b \qquad \varepsilon;b\equiv b$$

rules for recursion

$$\text{REC}\ bv.b\equiv \text{REC}\ bv'.b[bv\mapsto bv'] \qquad \text{where } bv'\notin FV(b)$$

$$\text{REC}\ bv.b\equiv b[bv\mapsto(\text{REC}\ bv.b)]$$

$$\frac{b\equiv b'}{\text{REC}\ bv.b\equiv \text{REC}\ bv.b'}$$

Fig. 5. Coercion rules for behaviours.

the theoretical development; the reason is that recursive behaviours are never explicitly introduced in the type inference system (Fig. 2).

3. Dynamic semantics of CML

We now present a structural operational semantics for the eager left-to-right evaluation of CML. The formulation is close in spirit to [25, 26] and amounts to the definition of three transition relations: one for *sequential* evaluation, one for *concurrent* evaluation and to handle the sync operator we also need a transition system for *matching* the communications against one another. One difference is that we add more book-keeping details[5] to the transition relations in order to be able to express a more informative subject reduction result (Propositions 4.6 and 4.9). Another difference is in the treatment of δ-reductions where we regard it too unrealistic to assume that all functions defined by δ-reduction have to be total. With respect to [1] the main difference is that [1] does not use the concept of evaluation context defined in [9] as a means of presenting operational semantics so as to lead to more "pleasant proofs" [33]; a minor difference is once again the exact choice of book-keeping details. So in summary

[5] These do *not* influence the sequence of configurations that evaluation passes through and hence could be dispensed with for the purposes of this section. They are merely added to strengthen the formulation of results or to facilitate shorter proofs.

$$E[\texttt{rec}\ i_0(i_1) : t_1 \to^b t_2 \Rightarrow e] \to E[(\texttt{fn}\ i_1 : t_1 \Rightarrow e)[(\texttt{rec}\ i_0(i_1) : t_1 \to^b t_2 \Rightarrow e)/i_0]]$$

$$E[(\texttt{fn}\ i : t \Rightarrow e)\ w] \to E[e[w/i]]$$

$$E[\texttt{let}\ i = w\ \texttt{in}\ e] \to E[e[w/i]]$$

$$E[\texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t] \to \begin{cases} E[e_1] & \text{if } w = \texttt{true} \\ E[e_2] & \text{if } w = \texttt{false} \end{cases}$$

$$E[w_1\, w_2] \to E[w_3] \quad \text{if } (w_1, w_2, w_3) \in \delta_{\to}$$

Fig. 6. Sequential evaluation.

we regard the close relationship between our semantics and those of [25, 26] as an indication of the soundness and generality of our approach.

3.1. Sequential evaluation

We begin with the sequential evaluation of expressions. This encompasses all features of CML *except* the `channel`, `fork` and `sync` primitives; these were the primitives listed in Fig. 1 that did not have an ε-behaviour associated with the function space. The definition of the transition relation is given in Fig. 6 and makes use of a number of auxiliary concepts. A central concept is that of an *evaluation context* E [9]. It may be defined inductively by:

$$E \ ::= \ [\,] \mid E\,e \mid w\,E \mid \texttt{let}\ i = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t$$

Here $[\,]$ is an empty context or a "hole"; so in general E describes an expression with precisely one hole in it. We then write $E[e]$ for the expression that is like E except that the hole is replaced by e. The definition of E is crucial for enforcing the left-to-right evaluation. As an example consider application, i.e. $e_1\, e_2$. The presence of $E\,e$ means that the function part, i.e. e_1, may always be evaluated whereas the presence of $w\,E$ means that the argument part, i.e. e_2, may only be evaluated after the function part has been evaluated (to a function abstraction or a constant).

Most of the axioms of Fig. 6 are now straightforward. The first axiom expresses the one-level unfolding of a recursive definition. For this we make use of the standard notation $e_1[e_2/i]$ for substituting e_2 for all free occurrences of i in e_1; when doing so care must of course be taken to rename bound identifiers in e_1 so as to avoid the capture of free identifiers in e_2. The second axiom is β-reduction and the use of w, rather than e, in the argument position is crucial for obtaining the call-by-value semantics. The third axiom is consistent with the view that $\texttt{let}\ i = e_1\ \texttt{in}\ e_2$ is semantically equivalent to $(\texttt{fn}\ i \Rightarrow e_2)\ e_1$. The fourth axiom is actually an abbreviation for two axioms describing the evaluation of the conditional depending on the outcome of the test.

The fifth axiom describes the δ-reductions for the constants of CML. The details are listed in Fig. 7 and once again make use of a number of auxiliary concepts. For a motivating example consider the intended reduction sequence:

$$\begin{aligned}&\texttt{pair (1+2) (3+4)} \rightarrow^{+} \texttt{pair 3 (3+4)} \rightarrow\\ &\langle\texttt{pair 3}\rangle\ \texttt{(3+4)} \rightarrow^{+} \langle\texttt{pair 3}\rangle\ \texttt{7} \rightarrow\\ &\langle\texttt{pair 3 7}\rangle\end{aligned}$$

Since our primitive functions may be curried and we want to distinguish between an expression like $\langle\texttt{pair 3}\rangle$ 7 and the corresponding value $\langle\texttt{pair 3 7}\rangle$, we extend the syntax of weakly evaluated expressions with the new "constants" $\langle\texttt{pair 3}\rangle$ and $\langle\texttt{pair 3 7}\rangle$. This is little more than a syntactic representation of the graphs used in the graph-based evaluation of functional languages [23].

Formally, we proceed as follows. Let c be one of the constants of Fig. 1 and let n be maximal such that TypeOf(c) may be written as $t'_1 \rightarrow^{\varepsilon} \cdots \rightarrow^{\varepsilon} t'_{n+1}$ where we have dispensed with the universal quantifiers. For each monotype instance $t = t_1 \rightarrow^{\varepsilon} \cdots \rightarrow^{\varepsilon} t_{n+1}$ of TypeOf(c) we then add the weakly evaluated constants $\langle c : t\, w_1\rangle, \ldots, \langle c : t\, w_1 \ldots w_n\rangle$ to the syntax of weakly evaluated expressions as indicated by

$$w \ ::= \ \ldots \mid \langle c : t\, w_1\rangle \mid \ldots \mid \langle c : t\, w_1 \ldots w_n\rangle$$

We also add a new typing rule:

$$\frac{tenv \vdash c : t \mid t_1 \rightarrow^{\varepsilon} \ldots \rightarrow^{\varepsilon} t_{n+1}\ \&\ \varepsilon \qquad tenv \vdash w_1 \mid t_1^{-}\ \&\ \varepsilon \quad \cdots \quad tenv \vdash w_i \mid t_i^{-}\ \&\ \varepsilon}{tenv \vdash \langle c : t_1 \rightarrow^{\varepsilon} \ldots \rightarrow^{\varepsilon} t_{n+1}\ w_1 \ldots w_i\rangle \mid t_{i+1} \rightarrow^{\varepsilon} \ldots \rightarrow^{\varepsilon} t_{n+1}\ \&\ \varepsilon}$$

where $i \leqslant n$ and $t_j^{-} \leqslant t_j$ for $j \leqslant i$

Returning to Fig. 7 most of the δ-"rules" are rather straightforward. A small point is that we deviate from [25] in not making a *meta-syntactic* distinction between weakly evaluated expressions of type t com b and those not of a type on this form; we simply use the meta-variable w whereas [25] uses meta-variables ev and v. More importantly we deviate from [25] in not requiring $\delta_{\rightarrow}$ to be total, e.g. we allow that we have no δ-"rule" for hd nil. We regard it overly restrictive to exclude this situation and instead introduce a new set $\delta_{\not\rightarrow}$ for characterizing these dynamically stuck[6] configurations. It may be defined by

$$(\texttt{hd} : t, \texttt{nil}) \in \delta_{\not\rightarrow} \quad \text{and} \quad (\texttt{tl} : t, \texttt{nil}) \in \delta_{\not\rightarrow}$$

and so allows us to distinguish between the situations (3+true)$\not\rightarrow$ that should have been caught by the type system and (hd nil) $\not\rightarrow$ that cannot be expected to be caught by any decidable type system.

[6] Alternatively, one could mask the dynamically stuck configurations using non-termination, e.g. to impose (hd: t, nil, hd: t nil) $\in \delta_{\rightarrow}$ as is essentially the approach of [18, Ch. 6].

pair: t	w_1	$\langle$pair: $t\ w_1\rangle$
$\langle$pair: $t\ w_1\rangle$	w_2	$\langle$pair: $t\ w_1\ w_2\rangle$
fst: t	$\langle$pair: $t'\ w_1\ w_2\rangle$	w_1
snd: t	$\langle$pair: $t'\ w_1\ w_2\rangle$	w_2
cons: t	w_1	$\langle$cons: $t\ w_1\rangle$
$\langle$cons: $t\ w_1\rangle$	w_2	$\langle$cons: $t\ w_1\ w_2\rangle$
hd: t	$\langle$cons: $t'\ w_1\ w_2\rangle$	w_1
tl: t	$\langle$cons: $t'\ w_1\ w_2\rangle$	w_2
isnil: t	nil	true
isnil: t	$\langle$cons: $t'\ w_1\ w_2\rangle$	false
send: t	w	$\langle$send: $t\ w\rangle$
receive: t	w	$\langle$receive: $t\ w\rangle$
choose: t	w	$\langle$choose: $t\ w\rangle$
wrap: t	w	$\langle$wrap: $t\ w\rangle$

Fig. 7. Tabulation of $(e_1, e_2, e_3) \in \delta_{\rightarrow}$.

3.2. Concurrent evaluation

The transition relation for concurrent evaluation is given in Fig. 8. Configurations have the form

$$cenv, PP$$

where *cenv* is a *channel environment* and *PP* is a *process pool*. More precisely, a process pool *PP* is a *partial function* from process identifiers $pi \in$ **PIdent** (e.g. p-0, p-1,...) to the expressions residing there. When writing a process pool PP' in the form $PP[pi_1 \mapsto e_1]\ldots[pi_n \mapsto e_n]$ we take it for granted that all of dom(PP), $\{pi_1\},\ldots,\{pi_n\}$ are mutually disjoint. The channel environment *cenv* is much like the type environment and so associates channel identifiers $ci \in$ **CIdent** (e.g. c-0, c-1, ...) with the type of values that may be communicated over the channel. We assume that the sets **Ident**, **PIdent** and **CIdent** are mutually disjoint. Also we formally regard a channel environment as a list of pairs of identifiers and types; as for the type environments we may then extract a partial function by mapping an identifier to the type of its rightmost occurrence. (The advantage of this view will only show up in later proofs.) The fact that we use a channel environment rather than just a set of

$$\frac{e \to e'}{cenv \,\&\, PP[pi \mapsto e] \Rightarrow^{\varepsilon}_{pi} cenv \,\&\, PP[pi \mapsto e']}$$

$$\frac{ci \notin \mathrm{dom}(cenv)}{cenv \,\&\, PP[pi \mapsto E[\texttt{channel} : (\texttt{unit} \to^{b} t)()]] \Rightarrow^{b}_{pi} cenv[ci \mapsto t] \,\&\, PP[pi \mapsto E[ci]]}$$

$$\frac{pi_2 \notin \mathrm{dom}(PP) \cup \{pi_1\}}{cenv \,\&\, PP[pi_1 \mapsto E[\texttt{fork} : (t \to^{b} \texttt{unit})\, w]] \Rightarrow^{b}_{pi_1, pi_2} cenv \,\&\, PP[pi_1 \mapsto E[()]][pi_2 \mapsto w()]}$$

$$\frac{(w_1, w_2) \rightsquigarrow (e_1, e_2) : (b_1, b_2)}{cenv \,\&\, PP[pi_1 \mapsto E_1[\texttt{sync} : t_1\, w_1]][pi_2 \mapsto E_2[\texttt{sync} : t_2\, w_2]] \Rightarrow^{b_1, b_2}_{pi_1, pi_2} cenv \,\&\, PP[pi_1 \mapsto E_1[e_1]][pi_2 \mapsto E_2[e_2]]}$$

Fig. 8. Concurrent Evaluation.

previously allocated channels, is an example of the book-keeping details[7] present in the semantics.

The first axiom embeds sequential evaluation within concurrent evaluation. There is no explicit mentioning of the evaluation context since this is all taken care of in Fig. 6. For book-keeping purposes the transition relation is labelled with the process evaluating and a *summary* of the communication behaviour; this will be useful in formulating and proving the results of the next section.

Next we have axioms for those primitives of Fig. 1 that were not dealt with in the definition of sequential evaluation. For channel allocation we use the channel environment to make sure that we do not re-allocate an already allocated channel. To record the allocation the channel environment is extended; for book-keeping purposes it turns out to be helpful for the next section that also the type is recorded and we do this by means of the channel environment. The behaviour labelling the arrow will be a monotype instance of $\forall\tau.\forall\iota.\ \tau$ CHAN ι.

The third axiom deals with process creation and is rather similar in spirit to the axiom for channel creation. The behaviour labelling the arrow will be a monotype instance of $\forall\tau.\forall\beta.\ \tau$ FORK β.

[7] If we were to use a set we would have to regard channel identifiers as constants, i.e. having an explicit type attached to each occurrence, and we would then need to formulate that all occurrences have the same type attached; this would turn out to be a more clumsy variation on the approach taken.

$$\langle \texttt{send:}\ (t_{11} \to^{\varepsilon} t_1\ \texttt{com}\ b_1)\langle \texttt{pair:}\ t_{12}\ ci\ w\rangle\rangle, \langle \texttt{receive:}\ (t_{21} \to^{\varepsilon} t_2\ \texttt{com}\ b_2)ci\rangle) \rightsquigarrow (w,w) : (b_1,b_2)$$

$$\frac{(w_1,w_3) \rightsquigarrow (e_1,e_3) : (b_1,b_3)}{(\langle \texttt{choose} : t_{11}\langle \texttt{cons} : t_{12}\, w_1\, w_2\rangle\rangle, w_3) \rightsquigarrow (e_1,e_3) : (b_1,b_3)}$$

$$\frac{(\langle \texttt{choose} : t_{11}\, w_2\rangle, w_3) \rightsquigarrow (e_2,e_3) : (b_2,b_3)}{(\langle \texttt{choose} : t_{11}\langle \texttt{cons} : t_{12}\, w_1\, w_2\rangle\rangle, w_3) \rightsquigarrow (e_2,e_3) : (b_2,b_3)}$$

$$\frac{(w_1,w_3) \rightsquigarrow (e_1,e_3) : (b_1,b_3)}{(\langle \texttt{wrap} : t_{11}\langle \texttt{pair} : t_{12}\, w_1\, w_2\rangle\rangle, w_3) \rightsquigarrow (w_2\, e_1, e_3) : (b_1,b_3)}$$

$$\frac{(w_1,w_2) \rightsquigarrow (e_1,e_2) : (b_1,b_2)}{(w_2,w_1) \rightsquigarrow (e_2,e_1) : (b_2,b_1)}$$

Fig. 9. Matching.

The fourth axiom takes care of communication among *different* processes. (That they are indeed different follows from the syntactic conventions mentioned above.) The formulation makes use of a transition system for expressing when two "delayed" communications match and for calculating the respective outcomes as well as indications of the communication behaviour. One of the behaviours labelling the arrow will be a monotype instance of $\forall\rho.\forall\tau.\ \rho!\tau$ and the other will be a monotype instance of $\forall\rho.\forall\tau.\ \rho?\tau$.

3.3. Matching

The transition system for matching is given in Fig. 9. Whereas the transition relations for sequential evaluation are small-step ones it is important that the transition relation for matching is a big-step one because this gives a kind of angelic non-determinism that ensures that "false matches" do not block the matching process.[8] The first axiom collects the values communicated between primitive `send` and `receive` constructs. The next two rules take care of the situation where the communication taking place in the first position amounts to choosing between several possibilities. The subsequent rule allows modifying the local version of the value communicated; it does not affect the value communicated as can be seen from the fact that only the value in one of the components is being modified and none of the behaviours have been changed. On top of this we would need the symmetric system of one axiom and three rules but to conserve space we follow [25] in "cheating" by adding the final non-structural "restructuring rule".

[8] In terms of implementation this is where the need for a kernel arises.

Remark. It would be possible to add additional rules to the transition system for concurrent evaluation. Assuming that there is some distinguished start process p-0 then the axiom

$$cenv \,\&\, PP[pi \mapsto w] \Rightarrow cenv \,\&\, PP \quad \text{if } pi \neq \texttt{p-0}$$

would describe the garbage collection of processes that have finished evaluation. In a similar way one could add an axiom for reclaiming channels no longer in use.

4. Deriving a process algebra from CML

We now show to which extent the types and behaviours are preserved or modified in the course of computation. This brings us to the main insight of this paper: a novel relation between (CML-like) programming languages and process algebras.

4.1. Sequential correctness

It is natural to restrict the attention to closed expressions, i.e. expressions with no free identifiers, because the definition of evaluation context is such that we never pass inside the scope of any defining occurrence for identifiers. However, we will have to allow that the expressions include channel identifiers that have been allocated in previous concurrent transitions. So we shall regard an expression e as being *closed* when $cenv \vdash e \mid t \,\&\, b$ for some *channel* environment *cenv*, type t and behaviour b. To make this consistent with the definition of Fig. 2 we shall allow *type* environments to range over program identifiers (**Ident**) as well as channel identifiers (**CIdent**).

Proposition 4.1. *If* $cenv \vdash e \mid t \,\&\, b$ *and* $e \rightarrow e'$ *then there exist* $t^- \leqslant t$ *and* $b^- \leqslant b$ *such that* $cenv \vdash e' \mid t^- \,\&\, b^-$.

Before approaching the proof it may be instructive to demonstrate why it would be too demanding to require that $t^- = t$ or $b^- = b$. For types suppose that $t^- < t$ is given and that $c : t^-$ is a constant; then $(\texttt{fn}\ x : t \Rightarrow x)(c : t^-)$ has type t but it evaluates to $c : t^-$ that has type t^-. For behaviours simply note that `if true then` e_1 `else` e_2 has behaviour $\varepsilon;(b_1 + b_2)$ and that it evaluates to e_1 that has behaviour b_1.

To conduct the proof we need several auxiliary results. The first lemma relates substitution to the use of the type environment. For the formulation recall that we regard type environments as lists of pairs (of identifiers and types) from which a partial function (from identifiers to types) can readily be recovered.

Lemma 4.2. *If* $i \notin dom(tenv_2)$, $cenv \vdash e_0 \mid t_0 \,\&\, \varepsilon$ *and* $cenv, tenv_1, [i \mapsto t_0], tenv_2 \vdash e \mid t \,\&\, b$ *then* $cenv, tenv_1, tenv_2 \vdash e[e_0/i] \mid t \,\&\, b$.

The proof is by induction on the typing inference for e and may be found in Appendix B. A simple consequence of this lemma is that if an identifier is not free in the expression then it may be removed from the type environment.

The second lemma may be read as saying that type and behaviour inference acts monotonically in the type environment as well as in the type and behaviour of subexpressions. To obtain a concise formulation we write $tenv_1 \leqslant tenv_2$ whenever $tenv_1$ and $tenv_2$ have equal length and pairs (i_1, t_1) and (i_2, t_2) in corresponding positions satisfy $i_1 = i_2$ and $t_1 \leqslant t_2$.

Lemma 4.3. *If* $cenv \vdash e_0 \mid t_0 \,\&\, b_0$, $cenv \vdash e_0' \mid t_0^- \,\&\, b_0^-$ *and* $cenv, tenv \vdash e[e_0/i] \mid t \,\&\, b$ *and also* $t_0^- \leqslant t_0, b_0^- \leqslant b_0$ *and* $tenv^- \leqslant tenv$; *then there exists* $t^- \leqslant t$ *and* $b^- \leqslant b$ *such that* $cenv, tenv^- \vdash e[e_0'/i] \mid t^- \,\&\, b^-$.

The proof is by induction on the syntax of the expression e and may be found in Appendix B. In a sense the lemma is two results in one and so we shall sometimes feel free to use the lemma without any substitution. An important consequence of the lemma is:

Corollary 4.4. *If* $cenv \vdash e_0 \mid t_0 \,\&\, b_0$, $cenv \vdash e_0' \mid t_0^- \,\&\, b_0^-$ *and* $cenv \vdash E[e_0] \mid t \,\&\, b$ *and also* $t_0^- \leqslant t_0$ *and* $b_0^- \leqslant b_0$; *then there exists* $t^- \leqslant t$ *and* $b^- \leqslant b$ *such that* $cenv \vdash E[e_0'] \mid t^- \,\&\, b^-$.

Proof. Simply use the fact that $E[e]$ equals $(E[i])[e/i]$ when i does not occur in E. □

One can now prove Proposition 4.1 by induction on the inference $e \to e'$; we refer to Appendix B for the details.

4.2. Matching correctness

The transition relation for concurrent evaluation utilizes the transition relations for sequential evaluation and for matching. It is therefore convenient to formulate the correctness of matching before considering the correctness of concurrent evaluation.

Proposition 4.5. *If* $cenv \vdash w_1 \mid (t_{01}\ \texttt{com}\ b_{01}) \,\&\, \varepsilon$, $cenv \vdash w_2 \mid (t_{02}\ \texttt{com}\ b_{02}) \,\&\, \varepsilon$ *and*

$$(w_1, w_2) \rightsquigarrow (e_1, e_2) : (b_1, b_2)$$

then there exists t_{01}^-, t_{02}^-, b_1' *and* b_2' *such that* $t_{01}^- \leqslant t_{01}$ *and* $t_{02}^- \leqslant t_{02}$ *and*

$$cenv \vdash e_1 \mid t_{01}^- \,\&\, b_1' \quad \textit{with } b_1; b_1' \leqslant b_{01},$$
$$cenv \vdash e_2 \mid t_{02}^- \,\&\, b_2' \quad \textit{with } b_2; b_2' \leqslant b_{02}$$

Furthermore, one of b_1 *and* b_2 *may be written* $r_1!t_1$ *and the other* $r_2?t_2$ *where* $t_1 \equiv t_2$ *and* r_1 *and* r_2 *have a lower bound:* $\exists r_0 : r_0 \leqslant r_1 \wedge r_0 \leqslant r_2$.

The proof is by induction on the transition relation for matching and may be found in Appendix B.

4.3. Concurrent correctness

So far we have not extended the notion of well-typing to the configurations of the concurrent transition relation and our first task is to remedy this. To this end we shall need a partial function PT of *process types*: it maps process identifiers $pi \in \mathbf{PIdent}$ to types. Similarly, we shall need a partial function PB of *process behaviours*: it maps process identifiers to behaviours. Intuitively, a process pool PP is correct with respect to PT and PB if each process, $PP(pi)$, has type and behaviour given by $PT(pi)$ and $PB(pi)$, respectively.

Formally, the correctness of PP with respect to PT and PB is written

$$\vdash cenv, PP \mid PT \,\&\, PB$$

and is given by

$$\begin{aligned}&\mathrm{dom}(PP) = \mathrm{dom}(PT) = \mathrm{dom}(PB) \wedge \\ &\forall\, pi \in \mathrm{dom}(PP) :\ cenv \vdash PP(pi) \mid PT(pi) \,\&\, PB(pi)\end{aligned}$$

Our main results about concurrent evaluation are Propositions 4.6 and 4.9 that give information about the evolution of types and behaviours. A concise formulation requires some additional notation. We allow writing $\vec{b}$ for b as well as b_1, b_2 and similarly $\vec{pi}$ for pi as well as pi_1, pi_2. When writing $\{\vec{pi}\}$ this then stands for $\{pi\}$ or $\{pi_1, pi_2\}$, respectively. When $\mathscr{P}$ is a partial function from process identifiers we write $\mathscr{P}\backslash\{\vec{pi}\}$ for the restriction $\mathscr{P}\lceil(\mathrm{dom}(\mathscr{P})\backslash\{\vec{pi}\})$ of $\mathscr{P}$ to the subset $\mathrm{dom}(\mathscr{P})\backslash\{\vec{pi}\}$ of $\mathrm{dom}(\mathscr{P})$. This notation applies to process pools, process types and process behaviours. For process types PT and PT' we write $PT'[\vec{pi}] \leqslant PT[\vec{pi}]$ as an abbreviation for

$$\{\vec{pi}\} \subseteq \mathrm{dom}(PT') \wedge \forall pi \in \{\vec{pi}\} \cap \mathrm{dom}(PT) : PT'(pi) \leqslant PT(pi)$$

This definition takes care of the situation where new processes are created.

Proposition 4.6. *If* $\vdash cenv, PP \mid PT \,\&\, PB$ *and* $cenv, PP \Rightarrow^{\vec{b}}_{\vec{pi}} cenv', PP'$ *then there exist* PT' *and* PB' *such that* $\vdash cenv', PP' \mid PT' \,\&\, PB'$ *and*

- $cenv' = cenv$ *unless* $\vec{b} = t_0$ CHAN i_0 *in which case* $cenv' = cenv[ci \mapsto t_0$ chan $i_0]$ *for some* $ci \notin dom(cenv)$,
- $PT'[\vec{pi}] \leqslant PT[\vec{pi}]$,
- *if* $\vec{b} = t_0$ FORK b_0 *and* $\vec{pi} = pi_1, pi_2$ *then* $PT'(pi_2) \leqslant t_0$

and where no changes take place outside $\{\vec{pi}\}$: $PP'\backslash\{\vec{pi}\} = PP\backslash\{\vec{pi}\}$, $PT'\backslash\{\vec{pi}\} = PT\backslash\{\vec{pi}\}$, *and* $PB'\backslash\{\vec{pi}\} = PB\backslash\{\vec{pi}\}$.

In Proposition 4.9 we shall strengthen this result by stating a stronger relationship between PB and PB'. The proof of Proposition 4.6 is by cases on the rule used for the concurrent transition and may be found in Appendix B. It makes use of the following generalization of Corollary 4.4:

Lemma 4.7. *If* $cenv \vdash e_0 \mid t_0 \,\&\, b_0$, $cenv \vdash e_0' \mid t_0' \,\&\, b_0'$, $cenv \vdash E[e_0] \mid t \,\&\, b$ *and also* $t_0' \leqslant t_0$ *and* $b^\bullet; b_0' \leqslant b_0$; *then there exist* t' *and* b' *such that* $cenv \vdash E[e_0'] \mid t' \,\&\, b'$ *and also* $t' \leqslant t$ *and* $b^\bullet; b' \leqslant b$.

The proof is by induction on the structure of E and may be found in Appendix B.

4.4. *The process algebra*

The statement of Proposition 4.6 (as opposed to its proof) does not convey much information about the relationship between $PB[\vec{pi}], \vec{b}$ and $PB'[\vec{pi}]$. This will be rectified now and our main tools will be two transition relations: one for the evolution of individual behaviours and one for the evolution of process behaviours.

The transition relation for individual behaviours takes the form

$$b_1 \longmapsto^a b_2$$

and says that the behaviour $b_1 \in \mathbf{Beh}$ evolves to $b_2 \in \mathbf{Beh}$ while performing the action a. It is possible to equate actions and behaviours but it is more informative to be more restrictive. To this end we define actions $a \in \mathbf{Act}$ by:

$$a \quad ::= \quad \varepsilon \mid r!t \mid r?t \mid t \,\mathrm{CHAN}\, r \mid t \,\mathrm{FORK}\, b$$

The details of the transition system are given in Fig. 10. The first axiom simply records the effect of performing an individual action. Then we have a rule that allows evolution of actions to take place in more elaborate contexts. The next rule is patterned after a "structural" rule

$$\frac{b_1 \equiv b_1' \qquad b_1' \longmapsto^a b_2' \qquad b_2' \equiv b_2}{b_1 \longmapsto^a b_2}$$

as might be found in the π-calculus [15]. However, because of our use of subtyping we find that we need a stronger rule and to obtain this we replace $\equiv$ by $\longmapsto^\varepsilon$ and add three more axioms. The first says that $\equiv$ is contained in $\longmapsto^\varepsilon$ and the final two[9] allow us to discard possible behaviours.

It is important that we have:

Lemma 4.8. *The statement* $b_1 \longmapsto^a b_2$ *is equivalent to the statement* $a; b_2 \leqslant b_1$.

Proof. The implication from left to right may be established by induction on the inference tree for $b_1 \longmapsto^a b_2$: that $a; b_2 \leqslant b_1$ holds is clear for the axioms and is maintained by the rules. For the converse implication assume that $a; b_2 \leqslant b_1$. It follows that $(a; b_2) + b_1 \equiv b_1$ and hence $b_1 \longmapsto^\varepsilon a; b_2$. From $a \longmapsto^a \varepsilon$ we next get $a; b_2 \longmapsto^a \varepsilon; b_2$ and since $\varepsilon; b_2 \equiv b_2$ we then have $\varepsilon; b_2 \longmapsto^\varepsilon b_2$. Putting this together we have $b_1 \longmapsto^a b_2$ as desired.

[9] Actually $b_1 + b_2 \longmapsto^\varepsilon b_2$ is derivable from the remaining axioms and rules.

$$a \longmapsto^{a} \varepsilon$$

$$\frac{b_1 \longmapsto^{a} b_2}{b_1; b \longmapsto^{a} b_2; b}$$

$$\frac{b_1 \longmapsto^{\varepsilon} b_1' \qquad b_1' \longmapsto^{a} b_2' \qquad b_2' \longmapsto^{\varepsilon} b_2}{b_1 \longmapsto^{a} b_2}$$

$$b \longmapsto^{\varepsilon} b' \quad \text{if } b' \equiv b$$

$$b_1 + b_2 \longmapsto^{\varepsilon} b_1$$

$$b_1 + b_2 \longmapsto^{\varepsilon} b_2$$

Fig. 10. Evolution of behaviours.

$$\frac{b \longmapsto^{\varepsilon} b'}{PB[pi \mapsto b] \Longrightarrow^{\varepsilon}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \longmapsto^{t\,\mathrm{CHAN}\,r} b'}{PB[pi \mapsto b] \Longrightarrow^{t\,\mathrm{CHAN}\,r}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \longmapsto^{t\,\mathrm{FORK}\,b_0} b' \qquad pi' \notin \mathrm{dom}(PB) \cup \{pi\} \qquad b_0 \longmapsto^{\varepsilon} b_0'}{PB[pi \mapsto b] \Longrightarrow^{t\,\mathrm{FORK}\,b_0}_{pi,pi'} PB[pi \mapsto b'][pi' \mapsto b_0']}$$

$$\frac{b_1 \longmapsto^{r_1!t_1} b_1' \quad b_2 \longmapsto^{r_2?t_2} b_2' \quad t_1 \equiv t_2 \qquad \exists r_0 : r_1 \geqslant r_0 \leqslant r_2}{PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow^{r_1!t_1, r_2?t_2}_{pi_1, pi_2} PB[pi_1 \mapsto b_1'][pi_2 \mapsto b_2']}$$

Fig. 11. Evolution of process behaviours.

The transition relation for process behaviours takes the form

$$PB \Longrightarrow^{\vec{b}}_{\vec{pi}} PB'$$

and says that the process behaviour PB evolves to the process behaviour PB'. Regarding process behaviours as a process algebra this transition relation then gives the operational semantics of terms in the process algebra. The details of the transition system are given in Fig. 11 and make use of the transition relation for individual behaviours.

Our main result linking CML with Behaviours to the process algebra is the following extension of Proposition 4.6:

Proposition 4.9. *Under the assumptions of Proposition* 4.6 *we may additionally conclude that*

- $PB \Longrightarrow^{\vec{b}}_{\vec{pi}} PB'$

The proof simply amounts to inspecting the proof of Proposition 4.6 and checking that the process behaviour PB' constructed there satisfies the new claim; for this Lemma 4.8 is most useful.

4.5. *On the role of structural equivalences*

The semantics of the π-calculus [15] was formulated in a two-step manner: first some structural laws are defined and then the transition relation (or some similar notion) is defined. The important point is that the structural laws are *included* in the definition of the transition relation (as shown in the "structural" rule). To motivate this approach Milner [15, Section 5.2] writes: "The reader who is familiar with [$\cdots$] will notice how much simpler our operational semantics has become. Of course, some of the complexity is concealed in the laws of structural congruence; but those laws are so to speak digestible without concern for the dynamics of action, and therefore deserve to be factored apart from the dynamics".

Recent papers have followed this trend. One example is [7] and, since it is not important whether one focuses on an equivalence or a preorder, also [19] and the present paper may be so regarded. However, a disadvantage of this approach is that it becomes hard (perhaps impossible) to argue for the adequacy of the selection of laws. Our systematic way of listing/selecting the axioms and inference rules was devised so as to avoid the reader getting the impression that the selection was arbitrary or contained "incorrect" axioms and rules. Once the language grows in complexity, as in [7], it is easy to get such impressions. A good example of the problems is the potential incorporation of the contraction rule

$$\frac{b_1 \leqslant b[bv \mapsto b_1] \qquad b[bv \mapsto b_2] \leqslant b_2}{b_1 \leqslant b_2}$$

if b guards bv and bv is positive in b

that would allow us to reason about recursive behaviours (see Appendix A). Its inclusion requires "correct" definitions of the notions of "positive" and "guarded".

One would have hoped that semantics could help in determining the "correctness" of the axioms, rules and side-conditions rather than having to rely on systematic ways of listing/selecting axioms and inference rules. Indeed the notion of observational equivalence is often used to show the soundness of the axiomatization. But observational equivalence is based on the semantics that is again based on the axioms, rules and side-conditions; trying to use this to argue for the choice of axioms, rules and side-conditions then becomes a circular argument!

Interestingly [27] departs from [15] in only incorporating structural laws corresponding to α-renaming (i.e. renaming bound variables). Then several equivalences are studied and most of the structural laws of [15] are proved to be sound. Once this result has been established one can revert to the use of the "structural" rule. The development of [27] therefore supports our belief that it may be dangerous to follow [15] in defining semantics by first stating a non-trivial "structural equivalence". Rather one should

define a more traditional operational semantics and then use simulation to validate the axioms and rules; this is what [27] does for the π-calculus and what [20] does for behaviours. Based on our experience this is the approach to adopt for future work in this area.

5. Conclusion

We started our work with an existing programming language. The first step was to "extend the *type* system" with additional information about the communication phenomena that take place during evaluation; rather than merely extending the syntax of types we introduced the notion of *behaviours* based on the concept of effect system [13]. The second step was to annotate the standard operational semantics in such a way that behaviours and types were properly propagated but without influencing the semantics. The third step then was to prove this formally by means of "subject reduction" results. To do so the behaviours were equipped with an operational semantics and in this way turned into a process algebra [11, 14].

The concept of types is well-known: a type talks about a set of values upon which the program operates. The types may be brought to life by interpreting the type constructors (e.g. product and function space) as logical connectives (e.g. conjunction and implication) in the manner popularized under the "propositions as types" slogan. Our approach performs a similar development for behaviours (corresponding to types) and computations (corresponding to values): a behaviour talks about certain aspects of the communications taking place during evaluation. And behaviours may be brought to life by being equipped with an operational semantics. Its relationship to the operational semantics of the underlying programming language merely amounts to a "subject reduction" result; this then is where our novel relation between programming languages and process algebras manifests itself.

The underlying distinction between types and behaviours that we have focused on is well-known in database theory: static constraints talk about the consistent states of the database (for example that one temperature is always less than another) whereas dynamic constraints talk about the consistent changes to the database (for example that a certain temperature never increases). We believe that the notion of types is the mathematical concept that formalizes static constraints and that similarly the notion of behaviours is the mathematical concept that formalizes dynamic constraints.

The relationship between programs and types is often approached in the following way: start with an untyped programming languages and its operational semantics. Then introduce the notion of types such that "well-typed programs do not go wrong" and there exists (sound and complete) type inference algorithms. Thus the program is primary and the types are secondary (although reliable programming calls for developing programs in typed languages). Our approach to behaviours is exactly analogous: introduce the notion of behaviours such that programs only evaluate in ways accounted for by the behaviours and such that behaviour inference algorithms exist. While we

have not dealt with behaviour inference algorithms here this is the subject of [21]: by dispensing with "subbehaviours" we may introduce polymorphism [20] and then obtain a sound behaviour inference algorithm [10]. (Integrating polymorphism with "subbehaviours" is still open.)

The relationship between types and programs is the non-trivial content of the "Curry–Howard isomorphism" within the "propositions as types" slogan. In this paper we have demonstrated that a notion of behaviours relates to programs in an analogous way. This is philosophically quite different from merely translating programs into behaviours of a process algebra in order to *define* the semantics of the program [14]. We may capitalize on this insight by claiming that the slogan "propositions as types" generalizes to "processes as behaviours".

Appendix A. Variations on the subtyping

The main idea behind the ordering $t_1 \leqslant t_2$ on types is that t_2 is more permissive than t_1 in the communications being allowed but that the "underlying" types t_1 and t_2 must be equal. There is scope for some variation here corresponding to the possibility of imposing additional (or fewer) axioms and rules for behaviours. We believe that the theoretical development of this paper is fairly robust to *extensions* of the set of rules and axioms in that the results proved still hold.

As a simple example one might contemplate adding the distributive laws

$$(r_1 + r_2)!t \equiv (r_1 !t) + (r_2 !t), \qquad (r_1 + r_2)?t \equiv (r_1 ?t) + (r_2 ?t),$$
$$t \text{ CHAN } (r_1 + r_2) \equiv (t \text{ CHAN } r_1) + (t \text{ CHAN } r_2),$$
$$b;(b_1 + b_2) \equiv (b;b_1) + (b;b_2), \qquad (b_1 + b_2);b \equiv (b_1;b) + (b_2;b)$$

This would not invalidate the theoretical development (even though the axiom $b;(b_1 + b_2) \equiv (b;b_1) + (b;b_2)$ "conflicts" with the CCS view of observational equivalence).

We now illustrate two more interesting examples of why to impose additional axioms and rules.

A.1. Interfacing with modules

Consider the following fragment of a program:

```
(fn f : unit →^{b_f} unit ⇒ ···)
    (rec g x : unit →^{b_g} unit ⇒
    let y = sync (receive ach)
    in let z = sync (send (pair bch y))
       in if ··· then g () else ())
```

[10] This work is based on the alternative approach to operational semantics discussed in Section 4.

We shall assume that ach has type `int chan a` and bch has type `int chan b`. Here g is a concrete program that a number of times will receive an integer over ach and then retransmit it over bch. Its type is `g : unit` $\rightarrow^{b_g}$ `unit` where

$$b_g = \text{REC}\beta.\ \texttt{a?int};\ \texttt{b!int};\ (\beta + \varepsilon)$$

Similarly (`fn f:unit` $\rightarrow^{b_f}$ `unit` $\Rightarrow \cdots$) is some module that requires that the argument obeys a certain protocol. This protocol says that the only communications allowed are the input of integers over some channel in region a and the output of integers over some channel in region b. This may be described more formally by

$$b_f = \text{REC}\beta.\ (a?\texttt{int};\beta) + (b!\texttt{int};\beta) + \varepsilon$$

We would then like to show that

$$\texttt{unit} \rightarrow^{b_g} \texttt{unit} \leqslant \texttt{unit} \rightarrow^{b_f} \texttt{unit}$$

corresponding to the fact that g obeys the protocol of f. Using the rules of Fig. 4 this amounts to showing $b_g \leqslant b_f$.

Unfortunately the axioms and rules of Fig. 5 do not suffice for proving $b_g \leqslant b_f$. This suggests adding a "contraction rule"

$$\frac{b_1 \leqslant b[bv \mapsto b_1] \qquad b[bv \mapsto b_2] \leqslant b_2}{b_1 \leqslant b_2}$$

if b guards bv and bv is positive in b.

This generalizes a rule from [4] and is explained in the sequel. A behaviour variable is *positive* in a behaviour if all occurrences have positive polarity in the sense of Section 3. This is a standard concept and we shall not go deeper into it here although some care is called for due to the presence of recursion. Intuitively, a behaviour b *guards* a behaviour variable bv if each "path" from the "beginning" of b to bv must pass through a non-empty behaviour. The precise details are subtle because bv does not occur guarded in $b = b'; bv$ if we have $b'; b' \equiv b'$; this incorporates the "obvious" $b' = \varepsilon$ and the not so obvious $b' = \text{REC}\beta.\ \beta; \beta$. We shall not go further into this here but only show that with the above rule we can now show $b_g \leqslant b_f$ where b_g and b_f are as above.

For this define

$$b\ =\ \texttt{a?int;b!int;}(\beta + \varepsilon)$$

Since $b_g = \text{REC}\beta.b$ it is evident that

$$b_g \leqslant b[\beta \mapsto b_g]$$

using the axiom for the one-level onfolding of REC. Although we have not defined "positive" and "guards" formally it should be immediate that β is "positive" in b and that b "guards" β. Using the contraction rule above it then suffices to show

$$b[\beta \mapsto b_f] \leqslant b_f$$

To do so we calculate

$$\begin{aligned} b[\beta \mapsto b_f] &\equiv \mathtt{a?int;b!int;}(b_f + \varepsilon) \\ &\equiv \mathtt{a?int;b!int;}(\mathtt{a?int;}b_f + \mathtt{b!int;}b_f + \varepsilon + \varepsilon) \\ &\equiv \mathtt{a?int;b!int;}b_f \\ &\leqslant \mathtt{a?int;}(\mathtt{a?int;}b_f + \mathtt{b!int;}b_f + \varepsilon) \\ &\equiv \mathtt{a?int;}b_f \\ &\leqslant \mathtt{a?int;}b_f + \mathtt{b!int;}b_f + \varepsilon \\ &\equiv b_f \end{aligned}$$

and the result follows.

By contrast, if $b'_g = \mathtt{a?int;b!int}$ and we were to show $b'_g \leqslant b_f$ then no contraction rule would be needed: just use the axiom for the one-level unfolding of REC twice and then some simple axioms.

A.2. Coarsening the structure

Early on we said that we intended to deviate from [17] in keeping the dependencies between individual communications. However, suppose that now we want to coarsen the structure of behaviours so that these distinctions no longer are made. One possibility is to add the axioms

$$b_1 + b_2 \equiv b_1; b_2, \qquad \text{REC } bv.\ b \equiv b, \qquad bv \equiv \varepsilon$$

The first expresses that we no longer distinguish between choice and sequencing. The next two axioms have the effect of removing the "REC bv." binder as well as behaviour variables; for closed behaviours this would be equivalent to the axiom REC $bv.\ b \equiv b[bv \mapsto \varepsilon]$.

An alternative presentation of the same idea is to translate the behaviours $b \in \mathbf{Beh}$ to a simpler structure of behaviours $b' \in \mathbf{Beh}'$ given by:

$$b' ::= \varepsilon \mid r!t \mid r?t \mid t \text{ CHAN } r \mid t \text{ FORK } b' \mid b'_1 \cup b'_2$$

Here $(b_1 + b_2)' = (b_1; b_2)' = b'_1 \cup b'_2$, $(\text{REC } bv.\ b)' = b'$ and $bv' = \varepsilon$. Comparing with the approach of [17] we have now lost the dependency between communications and $b'_1 \cup b'_2$ expresses that each of b'_1 and b'_2 may be performed zero, one or many times and in arbitrary order. This is the same interpretation of the union operator $\cup$ as in [17]. Interestingly we are now close to the formulations of [3, 32].

However, we still deviate from [17] in keeping behaviours and effects separate. To mimic the development in [17] more closely we may translate types $t \in \mathbf{Type}$ and behaviours $b \in \mathbf{Beh}$ into the so-called behaviour types $[\![t]\!]$, $[\![b]\!]$, $bt \in \mathbf{BehTyp}$ given by

$$\begin{aligned} bt ::= {} & \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \mid bt_1 \times bt_2 \mid bt\ \mathtt{list} \\ & \mid bt_1 \to bt_2 \mid bt\ \mathtt{chan}\ r \mid \varepsilon \mid r!bt \mid r?bt \\ & \mid \mathtt{fork}\ bt \mid bt_1 \cup bt_2 \end{aligned}$$

Most translations are fairly simple structural definitions. Some of the more interesting ones are:

$$\begin{aligned}
[\![t_1 \times t_2]\!] &\equiv [\![t_1]\!] \times [\![t_2]\!] \\
[\![t_1 \to^b t_2]\!] &\equiv [\![b]\!] \cup ([\![t_1]\!] \to [\![t_2]\!]) \\
[\![t \texttt{ chan } r]\!] &\equiv [\![t]\!] \texttt{ chan } r \\
[\![t \texttt{ com } b]\!] &\equiv [\![t]\!] \cup [\![b]\!] \\
[\![r!t]\!] &\equiv r![\![t]\!] \\
[\![t \text{ CHAN } r]\!] &\equiv [\![t]\!] \texttt{ chan } r \\
[\![t \text{ FORK } b]\!] &\equiv \texttt{fork } ([\![t]\!] \cup [\![b]\!]) \\
[\![b_1; b_2]\!] &\equiv [\![b_1]\!] \cup [\![b_2]\!]
\end{aligned}$$

The resulting system is pretty close in spirit to [17] and the remaining differences are due to the differences between the underlying languages (CML versus TPL [17]).

Appendix B. Proofs of main results

B.1. Proof of Lemma 4.2

We proceed by induction on the typing inference for e. So we must consider each axiom and rule of Fig. 2 as well as the rule added in Section 3.

Constants. Then e is a constant and $e[e_0/i]$ equals e. The result is then immediate.

Identifiers. Then we have two cases. If e is an identifier different from i the result follows as for constants. If e is identical to i then $t = t_0$ and $b = \varepsilon$. But $e[e_0/i]$ equals e_0 and by assumption $cenv \vdash e_0 \mid t \,\&\, b$. To obtain the desired result we modify the proof tree for $cenv \vdash e_0 \mid t \,\&\, b$ as follows: each node must be of the form [11]

$$cenv, tenv \vdash e_1 \mid t_1 \,\&\, b_1$$

and we replace it by

$$cenv, tenv_1, tenv_2, tenv \vdash e_1 \mid t_1 \,\&\, b_1$$

obtaining the desired proof tree for $cenv, tenv_1, tenv_2 \vdash e_0 \mid t \,\&\, b$.

Abstraction. Then e must be of the form $\texttt{fn } i_1 : t_1 \Rightarrow e_1$ and $t = t_1 \to^{b_1} t_2$ and $b = \varepsilon$. We have two cases. If i is different from i_1 then the induction hypothesis is applicable to

$$cenv, tenv_1, [i \mapsto t_0], tenv_2, [i_1 \mapsto t_1] \vdash e_1 \mid t_2 \,\&\, b_1$$

[11] Here we make use of the fact that type environments are lists rather than functions.

and yields $cenv, tenv_1, tenv_2, [i_1 \mapsto t_1] \vdash e_1[e_0/i] \mid t_2 \,\&\, b_1$ from which the desired

$$cenv, tenv_1, tenv_2 \vdash e[e_0/i] \mid t \,\&\, b$$

follows because $e[e_0/i]$ equals $\texttt{fn}\ i_1 : t_1 \Rightarrow (e_1[e_0/i])$.

The second case is when i is identical to i_1. Then $e[e_0/i]$ equals e and from

$$cenv, tenv_1, [i \mapsto t_0], tenv_2 \vdash e \mid t \,\&\, b \qquad (\star)$$

we must infer $cenv, tenv_1, tenv_2 \vdash e \mid t \,\&\, b$. We do this by modifying the proof tree for $(\star)$ as follows: each node must be of the form

$$cenv, tenv_1, [i \mapsto t_0], tenv_2, tenv \vdash e_2 \mid t_3 \,\&\, b_2$$

and we replace it by

$$cenv, tenv_1, tenv_2, tenv \vdash e_2 \mid t_3 \,\&\, b_2.$$

This is valid since $i \in \mathrm{dom}(tenv)$ whenever i is free in e_2.

Application. Then e must be of the form $e_1\, e_2$. Inspecting the proof tree we must have premisses of the form

$$cenv, tenv_1, [i \mapsto t_0], tenv_2 \vdash e_1 \mid t_1 \rightarrow^{b_3} t \,\&\, b_1$$
$$cenv, tenv_1, [i \mapsto t_0], tenv_2 \vdash e_2 \mid t_1^- \,\&\, b_2$$

with $t_1^- \leqslant t_1$ and $b = b_1; b_2; b_3$. The induction hypothesis is applicable and yields

$$cenv, tenv_1, tenv_2 \vdash e_1[t_0/i] \mid t_1 \rightarrow^{b_3} t \,\&\, b_1$$
$$cenv, tenv_1, tenv_2 \vdash e_2[t_0/i] \mid t_1^- \,\&\, b_2$$

from which the desired result follows.

Let-abstraction. This case follows using combinations of the techniques from abstraction and application; this should not be surprising because in the absence of polymorphism the expression $(\texttt{fn}\ i_1 : t_1 \Rightarrow e_1)\, e_2$ is equivalent to $\texttt{let}\, i_1 = e_2\ \texttt{in}\ e_1$.

Recursion. Then e must be of the form $\texttt{rec}\ i_0(i_1) : t \Rightarrow e_1$ and $t = t_1 \rightarrow^{b_1} t_2$ and $b = \varepsilon$. If i_0, i_1 and i are all different we proceed as follows. Inspection of the proof tree reveals a premiss of the form

$$cenv, tenv_1, [i \mapsto t_0], tenv_2, [i_0 \mapsto t][i_1 \mapsto t_1] \vdash e_1 \mid t_2^- \,\&\, b_1^-$$

where $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1$. Applying the induction hypothesis we obtain

$$cenv, tenv_1, tenv_2, [i_0 \mapsto t][i_1 \mapsto t_1] \vdash e_1[e_0/i] \mid t_2^- \,\&\, b_1^-$$

and this yields

$$cenv, tenv_1, tenv_2 \vdash \texttt{rec}\, i_0(i_1) : t \Rightarrow e_1[e_0/i] \mid t \,\&\, b$$

which is indeed the desired result.

If i equals one or more of i_0 and i_1 then $e[e_0/i]$ equals e. We then prove the desired result as in the similar case for abstraction.

Conditional. This case follows using the techniques from application; this should not be surprising because for the purposes of type inference the expression `if` e' `then` e_1 `else` $e_2 : t$ behaves as the nested application $(\texttt{cond}: t')\, e'\, e_1\, e_2$.

Weakly evaluated constants. Given our decision to take $w ::= i$ rather than $e ::= i$ in the syntax this is not a trivial case because identifiers may occur in weakly evaluated expressions of the form $\langle c\ w_1 \ldots w_m \rangle$. However, the proof may be conducted using the techniques from application. □

B.2. Proof of Lemma 4.3

We proceed by induction on the syntax of the expression e; this includes the syntactic category of weakly evaluated expressions.

Constants. Then e is of the form $c : t$ and $b = \varepsilon$. It follows that $e[e_0/i]$ and $e[e_0'/i]$ both equal $c : t$ and the result is then straightforward: let $t^- = t$ and $b^- = b$.

Identifiers. We have two cases. If the expression e is different from the identifier i then $e[e_0/i]$ and $e[e_0'/i]$ both equal e and we have $t = tenv(i)$ and $b = \varepsilon$. By taking $t^- = tenv^-(i)$ and $b^- = b$ the result is then straightforward.

The other case is when e is identical to i. Then $e[e_0/i]$ equals e_0. From the assumption

$$cenv \vdash e_0 \mid t_0 \,\&\, b_0$$

we may obtain $cenv, tenv \vdash e_0 \mid t_0 \,\&\, b_0$ by modifying the proof tree in the manner demonstrated in the proof of Lemma 4.2. Since we also have

$$cenv, tenv \vdash e_0 \mid t \,\&\, b$$

it follows from Fact 2.2 that $t = t_0$ and $b = b_0$. Since $e[e_0'/i]$ equals e_0' we obtain

$$cenv, tenv \vdash e[e_0'/i] \mid t_0^- \,\&\, b_0^-$$

by performing a simple modification of the proof tree. Taking $t^- = t_0^-$ and $b^- = b_0^-$ then gives the desired result.

Abstraction. Then e must be of the form $\texttt{fn}\ i_1 : t_1 \Rightarrow e_1$ and $t = t_1 \rightarrow^{b_1} t_2$ and $b = \varepsilon$. We have two cases. If i is identical to i_1 then both $e[e_0/i]$ and $e[e_0'/i]$ equal e. Inspection of the proof tree for $cenv, tenv \vdash e \mid t \,\&\, b$ then reveals a premiss

$$cenv, tenv[i_1 \mapsto t_1] \vdash e_1 \mid t_2 \,\&\, b_1$$

The induction hypothesis is applicable because e_1 equals $e_1[e_0/j]$ for a fresh identifier j and yields

$$cenv, tenv^-[i_1 \mapsto t_1] \vdash e_1 \mid t_2^- \,\&\, b_1^-$$

for $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1$. It follows that

$$cenv, tenv^- \vdash e \mid t^- \,\&\, b^-$$

with $t^- = t_1 \rightarrow^{b_1^-} t_2^-$ and $b^- = \varepsilon$.

The other case is when i is different from i_1. Inspection of the proof tree for $cenv, tenv \vdash e[e_0/i] \mid t \,\&\, b$ then reveals a premiss

$$cenv, tenv[i_1 \mapsto t_1] \vdash e_1[e_0/i] \mid t_2 \,\&\, b_1$$

The induction hypothesis is applicable and yields

$$cenv, tenv^-[i_1 \mapsto t_1] \vdash e_1[e_0'/i] \mid t_2^- \,\&\, b_1^-$$

for $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1$. It follows that

$$cenv, tenv^- \vdash e[e_0'/i] \mid t^- \,\&\, b^-$$

with $t^- = t_1 \rightarrow^{b_1^-} t_2^-$ and $b^- = \varepsilon$.

Application. Then e must be of the form $e_1\, e_2$. Inspecting the proof tree for $e[e_0/i]$ we find premisses of the form

$$cenv, tenv \vdash e_1[e_0/i] \mid t_1 \rightarrow^{b_3} t \,\&\, b_1, \qquad cenv, tenv \vdash e_2[e_0/i] \mid t_1^- \,\&\, b_2$$

with $t_1^- \leqslant t_1$ and $b = b_1; b_2; b_3$. The induction hypothesis is applicable and yields

$$cenv, tenv^- \vdash e_1[e_0'/i] \mid t_1^+ \rightarrow^{b_3^-} t^- \,\&\, b_1^-, \qquad cenv, tenv^- \vdash e_2[e_0'/i] \mid t_1^{--} \,\&\, b_2^-$$

for $t_1^+ \geqslant t_1$, $b_3^- \leqslant b_3$, $t^- \leqslant t$, $b_1^- \leqslant b_1$, $t_1^{--} \leqslant t_1^-$ and $b_2^- \leqslant b_2$. Since $t_1^{--} \leqslant t_1^+$ it follows that

$$cenv, tenv^- \vdash e[e_0'/i] \mid t^- \,\&\, b^-$$

with $b^- = b_1^-; b_2^-; b_3^-$. This shows the desired result.

Let-abstraction. Then e must be of the form `let` $j = e_1$ `in` e_2. As in the proof of Lemma 4.2 this case utilizes combinations of the techniques from abstraction and application. The additional complication is that the substitution into e_1 may present a new modification of the type environment for e_2. We have two cases. If the identifiers i and j are distinct then inspection of the proof tree for $e[e_0/i]$ reveals premisses of the form

$$cenv, tenv \vdash e_1[e_0/i] \mid t_1 \,\&\, b_1, \qquad cenv, tenv[j \mapsto t_1] \vdash e_2[e_0/i] \mid t \,\&\, b_2$$

with $b = b_1; b_2$. Applying the induction hypothesis to the first inference yields

$$cenv, tenv^- \vdash e_1[e_0'/i] \mid t_1^- \,\&\, b_1^-$$

where $t_1^- \leqslant t_1$ and $b_1^- \leqslant b_1$. Applying the induction hypothesis to the second inference yields

$$cenv, tenv^-[j \mapsto t_1^-] \vdash e_2[e_0'/i] \mid t^- \,\&\, b_2^-$$

with $t^- \leqslant t$. Taking $b^- = b_1^-; b_2^-$ this yields

$$cenv, tenv^- \vdash e[e_0'/i] \mid t^- \,\&\, b^-$$

which is the desired result.

The second case is when i and j are identical. As was illustrated for abstraction it is straightforward to modify the above proof so as to apply in this case: we still have to substitute in e_1 but should not do so in e_2.

Recursion. Then e must be of the form $\texttt{rec}\ i_0(i_1) : t \Rightarrow e_1$ and $t = t_1 \to^{b_1} t_2$ and $b = \varepsilon$. We have two cases. If i equals one or more of i_0 and i_1 then $e[e_0/i]$ and $e[e_0'/i]$ both equal e. Inspection of the proof tree for $e[e_0/i]$ then reveals a premiss

$$cenv, tenv[i_0 \mapsto t][i_1 \mapsto t_1] \vdash e_1 \mid t_2^- \,\&\, b_1^-$$

where $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1$. The induction hypothesis is applicable (because e_1 equals $e_1[e_0/j]$ for a fresh identifier j) and yields

$$cenv, tenv^-[i_0 \mapsto t][i_1 \mapsto t_1] \vdash e_1 \mid t_2^{--} \,\&\, b_1^{--}$$

for $t_2^{--} \leqslant t_2^-$ and $b_1^{--} \leqslant b_1^-$. It follows that

$$cenv, tenv^- \vdash e \mid t \,\&\, b$$

since $t_2^{--} \leqslant t_2$ and $b_1^{--} \leqslant b_1$.

The other case is when i_0, i_1 and i are all different. Inspection of the proof tree for $e[e_0/i]$ then reveals a premiss

$$cenv, tenv[i_0 \mapsto t][i_1 \mapsto t_1] \vdash e_1[e_0/i] \mid t_2^- \,\&\, b_1^-$$

where $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1$. The induction hypothesis is applicable and yields

$$cenv, tenv^-[i_0 \mapsto t][i_1 \mapsto t_1] \vdash e_1[e_0'/i] \mid t_2^{--} \,\&\, b_1^{--}$$

for $t_2^{--} \leqslant t_2^-$ and $b_1^{--} \leqslant b_1^-$. It follows that

$$cenv, tenv^- \vdash e[e_0'/i] \mid t \,\&\, b$$

because $e[e_0'/i]$ equals $\texttt{rec}\ i_0(i_1) : t \Rightarrow (e_1[e_0'/i])$.

Conditional. As in the proof of Lemma 4.2 this case follows using the techniques from application.

Weakly evaluated constants. As in the proof of Lemma 4.2 this case follows using the techniques from application. □

B.3. Proof of Proposition 4.1

We proceed by induction on the inference $e \to e'$, i.e. by case analysis according to Fig. 6.

Recursion. The inference $e \to e'$ must have the form

$$E[e_1] \to E[e_2[e_1/i_0]]$$

where e_1 and e_2 are given by

$$e_1 = \texttt{rec}\, i_0(i_1) : t_1 \to^{b_1} t_2 \Rightarrow e_0, \qquad e_2 = \texttt{fn}\, i_1 : t_1 \Rightarrow e_0$$

Since the proof tree for $cenv \vdash E[e_1] \mid t \,\&\, b$ follows the syntax of $E[e_1]$ it is possible to identify the node corresponding to the hole in E. It must have the form

$$cenv, tenv \vdash e_1 \mid t_1' \,\&\, b_1'$$

with *tenv* being empty because the definition of the evaluation context is such that the hole is never in the scope of any binding occurrence of an identifier. The premiss of this node must be

$$cenv[i_0 \mapsto t_1 \to^{b_1} t_2][i_1 \mapsto t_1] \vdash e_0 \mid t_2^- \,\&\, b_1^-$$

for $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1$. This then shows that $t_1' = t_1 \to^{b_1} t_2$ and $b_1' = \varepsilon$.

In the case where i_0 is distinct from i_1 we may use Lemma 4.2 to obtain a proof

$$cenv[i_1 \mapsto t_1] \vdash e_0[e_1/i_0] \mid t_2^- \,\&\, b_1^-$$

from which

$$cenv \vdash e_2[e_1/i_0] \mid t_1 \to^{b_1^-} t_2^- \,\&\, \varepsilon$$

is immediate. But by Corollary 4.4 and $t_1 \to^{b_1^-} t_2^- \leqslant t_1 \to^{b_1} t_2$ and $\varepsilon \leqslant b_1'$ there exist $t^- \leqslant t$ and $b^- \leqslant b$ such that

$$cenv \vdash E[e_2[e_1/i_0]] \mid t^- \,\&\, b^-$$

and this is the result.

In the case where i_0 is identical to i_1 we have that $e_2[e_1/i_0]$ equals e_2. From the typing of e_0 we may then obtain

$$cenv[i_0 \mapsto t_1 \to^{b_1} t_2] \vdash e_2 \mid t_1 \to^{b_1^-} t_2^- \,\&\, \varepsilon$$

and by straightforward modification of the proof tree we obtain

$$cenv \vdash e_2 \mid t_1 \to^{b_1^-} t_2^- \,\&\, \varepsilon$$

As before Corollary 4.4 then gives the desired result.

Application (β-reduction). The inference $e \to e'$ must have the form

$$E[(\texttt{fn}\ i : t_1 \Rightarrow e_0)w] \to E[e_0[w/i]]$$

Inspection of the proof tree for $cenv \vdash e \mid t \,\&\, b$ once more identifies a node

$$cenv \vdash (\texttt{fn}\ i : t_1 \Rightarrow e_0)\, w \mid t_2 \,\&\, b_1'$$

corresponding to the hole in E. It must have premisses

$$cenv \vdash (\texttt{fn}\ i : t_1 \Rightarrow e_0) \mid t_1 \rightarrow^{b_1} t_2 \,\&\, \varepsilon, \qquad cenv \vdash w \mid t_1^- \,\&\, b_2 \qquad (\star)$$

where $t_1^- \leqslant t_1$ and $b_1' = \varepsilon; b_2; b_1$. To see that $b_2 = \varepsilon$ we use:

Fact B.1. *If $cenv, tenv \vdash w \mid t \,\&\, b$ and w is a weakly evaluated expression then $b = \varepsilon$.*

Proof. A simple induction over weakly evaluated expressions. □

Proof of Proposition 4.1 (*continued*)

Inspection of the proof tree ($\star$) reveals a premiss

$$cenv[i \mapsto t_1] \vdash e_0 \mid t_2 \,\&\, b_1$$

and by Lemma 4.3 (with a trivial substitution) this yields

$$cenv[i \mapsto t_1^-] \vdash e_0 \mid t_2^- \,\&\, b_1^-$$

for $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1$. Using Lemma 4.2 we then obtain

$$cenv \vdash e_0[w/i] \mid t_2^- \,\&\, b_1^-$$

Since $t_2^- \leqslant t_2$ and $b_1^- \leqslant b_1 \leqslant \varepsilon; \varepsilon; b_1 \leqslant b_1'$ the desired result then follows from Corollary 4.4.

Let-abstraction. This case is slightly simpler than the one for application but we shall nonetheless provide the details. The inference $e \rightarrow e'$ must have the form

$$E[\texttt{let}\ i = w\ \texttt{in}\ e_0] \rightarrow E[e_0[w/i]]$$

Inspection of the proof tree for e once more identifies a node

$$cenv \vdash \texttt{let}\ i = w\ \texttt{in}\ e_0 \mid t_2 \,\&\, b_1'$$

corresponding to the hole in E. It must have premisses

$$cenv \vdash w \mid t_1 \,\&\, \varepsilon, \qquad cenv[i \mapsto t_1] \vdash e_0 \mid t_2 \,\&\, b_2$$

where $b_1' = \varepsilon; b_2$ and we have used Fact B.1. Using Lemma 4.2 we obtain

$$cenv \vdash e_0[w/i] \mid t_2 \,\&\, b_2$$

and since $t_2 \leqslant t_2$ and $b_2 \leqslant \varepsilon; b_2 \leqslant b_1'$ the desired result follows from Corollary 4.4.

Conditional. The inference $e \rightarrow e'$ must have the form

$$E[\texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t_0] \rightarrow E[e_1]$$

where, without loss of generality, we assume that w = true. Inspection of the proof tree for $cenv \vdash e \mid t \,\&\, b$ once more identifies a node

$$cenv \vdash \texttt{if } w \texttt{ then } e_1 \texttt{ else } e_2 : t_0 \mid t_0 \,\&\, b_0$$

corresponding to the hole in E. It must have premisses

$$cenv \vdash w \mid \texttt{bool} \,\&\, \varepsilon, \qquad cenv \vdash e_1 \mid t_1 \,\&\, b_1, \qquad cenv \vdash e_2 \mid t_2 \,\&\, b_2$$

where $t_1 \leqslant t_0$, $t_2 \leqslant t_0$, $b_0 = \varepsilon;(b_1 + b_2)$ and we have used Fact B.1. Applying Corollary 4.4 to $cenv \vdash e_1 \mid t_1 \,\&\, b_1$ we then get the desired result because $t_1 \leqslant t_0$ and $b_1 \leqslant b_1 + b_2 \leqslant \varepsilon;(b_1 + b_2) \leqslant b_0$.

Application (δ-*reduction*). The inference $e \to e'$ must have the form

$$E[w_1\, w_2] \to E[w_3]$$

where $(w_1, w_2, w_3) \in \delta_{\to}$(see Figure 7). Inspection of the proof tree for $cenv \vdash e \mid t \,\&\, b$ once more identifies a node

$$cenv \vdash w_1\, w_2 \mid t_2 \,\&\, b_0$$

corresponding to the hole in E. It must have premisses

$$cenv \vdash w_1 \mid t_1 \to^{b_1} t_2 \,\&\, \varepsilon, \qquad cenv \vdash w_2 \mid t_1^- \,\&\, \varepsilon$$

where $t_1^- \leqslant t_1$ and $b_0 = \varepsilon;\varepsilon;b_1$ and we have used Fact B.1. To obtain the desired result using Corollary 4.4 it suffices to find t_2^- and show

$$cenv \vdash w_3 \mid t_2^- \,\&\, \varepsilon, \qquad t_2^- \leqslant t_2, \qquad b_1 \equiv \varepsilon$$

as then $\varepsilon \leqslant \varepsilon;\varepsilon;b_1 \leqslant b_0$.

This may be achieved by inspection of Fig. 7. We only consider two typical cases. The case where

$$w_1 = \texttt{pair} : t_3 \to^{\varepsilon} t_4 \to^{\varepsilon} t_3 \times t_4, \qquad w_2 = w, \qquad w_3 = \langle w_1\, w_2 \rangle$$

is typical of the case where weakly evaluated expressions are constructed. In this case $t_1 = t_3$, $b_1 = \varepsilon$, $t_2 = t_4 \to^{\varepsilon} t_3 \times t_4$ and taking $t_2^- = t_2$ gives the desired result. The other case where

$$w_1 = \texttt{fst} : t_3 \times t_4 \to^{\varepsilon} t_3, \quad w_2 = \langle \texttt{pair} : t_3' \to^{\varepsilon} t_4' \to^{\varepsilon} t_3' \times t_4'\, w_a\, w_b \rangle, \quad w_3 = w_a$$

is typical of the case where weakly evaluated expressions are taken apart. In this case $t_1 = t_3 \times t_4$, $b_1 = \varepsilon$, $t_2 = t_3$, $t_1^- = t_3' \times t_4'$ and it follows that $t_3' \leqslant t_2$. From $cenv \vdash w_2 \mid t_1^- \,\&\, \varepsilon$ we then get $cenv \vdash w_a \mid t_3'^- \,\&\, \varepsilon$ where $t_3'^- \leqslant t_3'$ and we have used Fact B.1. Taking $t_2^- = t_3'^-$ then gives the desired result. □

B.4. Proof of Proposition 4.5

We proceed by induction on the transition relation for matching.

The axiom for send and receive. In this case

$$w_1 = \langle \texttt{send} : (t_{11} \rightarrow^{\varepsilon} t_1 \texttt{ com } b_1)\, \langle \texttt{pair} : t_{12}\, ci\, w \rangle \rangle$$

$$w_2 = \langle \texttt{receive} : (t_{21} \rightarrow^{\varepsilon} t_2 \texttt{ com } b_2)\, ci \rangle$$

where it is essential that the two occurrences of ci are indeed the same channel identifier. It is immediate from the typing rule for weakly evaluated constants that $t_1 = t_{01}$, $t_2 = t_{02}$, $b_1 = b_{01}$ and $b_2 = b_{02}$. Furthermore, b_1 may be written $r_1!t_1$ and b_2 may be written $r_2?t_2$. Finally, write $cenv(ci) = t_0 \texttt{ chan } r_0$.

Turning the attention to w and ci of w_1 it follows from

$$cenv \vdash w_1 \mid (t_{01} \texttt{ com } b_{01}) \,\&\, \varepsilon$$

that

$$cenv \vdash w \mid t_{01}^{-} \,\&\, \varepsilon, \qquad cenv \vdash ci \mid (t'_{01} \texttt{ chan } r_1^{-}) \,\&\, \varepsilon$$

for some $t_{01}^{-} \leqslant t_{01}$, $t'_{01} \equiv t_{01}$ and $r_1^{-} \leqslant r_1$. (The detailed argument observes that t_{11} must have the form $(t_1 \texttt{ chan } r_1) \times t_1$ with b_1 as above, and hence t_{12} must have the form $t_3 \rightarrow^{\varepsilon} t_4 \rightarrow^{\varepsilon} t_3 \times t_4$ with $t_3 \leqslant t_1 \texttt{ chan } r_1$ and $t_4 \leqslant t_1$, and hence $cenv \vdash ci \mid t_3^{-} \,\&\, \varepsilon$ and $cenv \vdash w \mid t_4^{-} \,\&\, \varepsilon$ for some $t_3^{-} \leqslant t_3$ and $t_4^{-} \leqslant t_4$.)

Turning the attention to ci of w_2 it follows from

$$cenv \vdash w_2 \mid (t_{02} \texttt{ com } b_{02}) \,\&\, \varepsilon$$

that

$$cenv \vdash ci \mid (t'_{02} \texttt{ chan } r_2^{-}) \,\&\, \varepsilon$$

for some $t'_{02} \equiv t_{02}$ and $r_2^{-} \leqslant r_2$. By the typing axiom for identifiers it follows that $t'_{01} = t_0 = t'_{02}$ and $r_1^{-} = r_0 = r_2^{-}$. A consequence of this is that $t_{01} \equiv t_{02}$ and hence $t_1 \equiv t_2$.

Taking $t_{02}^{-} = t_{01}^{-}$, $b'_1 = \varepsilon$ and $b'_2 = \varepsilon$ we then get

$$cenv \vdash w \mid t_{01}^{-} \,\&\, b'_1 \quad \text{with } b_1; b'_1 \leqslant b_{01}$$
$$cenv \vdash w \mid t_{02}^{-} \,\&\, b'_2 \quad \text{with } b_2; b'_2 \leqslant b_{02}$$

and we also have $t_{01}^{-} \leqslant t_{01}$ and $t_{02}^{-} \leqslant t_{02}$ (since $t_{02}^{-} = t_{01}^{-}$, $t_{01}^{-} \leqslant t_{01}$, $t_{01} \equiv t_{02}$). Furthermore, $b_1 = r_1!t_1$ and $b_2 = r_2?t_2$ with $t_1 \equiv t_2$ and $\exists r : r \leqslant r_1 \wedge r \leqslant r_2$.

The rule for choosing heads. In this case

$$w_1 = \langle \texttt{choose} : t_{11}\, \langle \texttt{cons} : t_{12}\, w_{11}\, w_{12} \rangle \rangle$$

It is immediate from the typing rule for weakly evaluated constants to infer from

$$cenv \vdash w_1 \mid (t_{01} \texttt{ com } b_{01}) \,\&\, \varepsilon$$

that

$$cenv \vdash w_{11} \mid (t'_{01}\,\mathtt{com}\; b'_{01})\&\,\varepsilon$$

for some $t'_{01} \leqslant t_{01}$ and $b'_{01} \leqslant b_{01}$.

By assumption

$$(w_{11}, w_2) \rightsquigarrow (e_1, e_2) : (b_1, b_2)$$

and the induction hypothesis gives $t'^{-}_{01} \leqslant t'_{01}$, $t^{-}_{02} \leqslant t_{02}$, b'_1 and b'_2 such that

$$cenv \vdash e_1 \mid t'^{-}_{01}\,\&\,b'_1 \quad \text{with } b_1; b'_1 \leqslant b'_{01}$$
$$cenv \vdash e_2 \mid t^{-}_{02}\,\&\,b'_2 \quad \text{with } b_2; b'_2 \leqslant b_{02}$$

By taking $t^{-}_{01} = t'^{-}_{01}$ it is immediate that

$$cenv \vdash e_1 \mid t^{-}_{01}\,\&\,b'_1 \quad \text{with } b_1; b'_1 \leqslant b_{01}$$
$$cenv \vdash e_2 \mid t^{-}_{02}\,\&\,b'_2 \quad \text{with } b_2; b'_2 \leqslant b_{02}$$

and that also $t^{-}_{01} \leqslant t_{01}$ and $t^{-}_{02} \leqslant t_{02}$.

The rule for choosing tails. In this case

$$w_1 = \langle \mathtt{choose} : t_{11}\, \langle \mathtt{cons} : t_{12}\, w_{11}\, w_{12}\rangle\rangle$$

and much as before

$$cenv \vdash \langle \mathtt{choose} : t_{11}\, w_{12}\rangle \mid (t_{01}\,\mathtt{com}\; b_{01})\&\,\varepsilon$$

(except that there is no need to consider $t'_{01} \leqslant t_{01}$ and $b'_{01} \leqslant b_{01}$). By assumption

$$(\langle \mathtt{choose} : t_{11}\, w_{12}\rangle, w_2) \rightsquigarrow (e_1, e_2) : (b_1, b_2)$$

and the induction hypothesis gives $t^{-}_{01} \leqslant t_{01}$, $t^{-}_{02} \leqslant t_{02}$, b'_1 and b'_2 such that

$$cenv \vdash e_1 \mid t^{-}_{01}\,\&\,b'_1 \quad \text{with } b_1; b'_1 \leqslant b_{01}$$
$$cenv \vdash e_2 \mid t^{-}_{02}\,\&\,b'_2 \quad \text{with } b_2; b'_2 \leqslant b_{02}$$

This is indeed the desired result.

The rule for wrap. In this case

$$w_1 = \langle \mathtt{wrap} : t_{11}\, \langle\, \mathtt{pair} : t_{12}\, w_{11}\, w_{12}\rangle\rangle$$

The type t_{11} must be of the form

$$t_{11} = (t_3\,\mathtt{com}\; b_3) \times (t_3 \rightarrow^{b_4} t_4) \rightarrow^{\varepsilon} (t_4\,\mathtt{com}\; (b_3; b_4))$$

and from $cenv \vdash w_1 \mid (t_{01}\,\mathtt{com}\; b_{01})\&\,\varepsilon$ it follows that $t_{01} = t_4$ and $b_{01} = b_3; b_4$. It further follows that

$$cenv \vdash w_{11} \mid (t^{-}_3\,\mathtt{com}\; b^{-}_3)\&\,\varepsilon, \qquad cenv \vdash w_{12} \mid (t^{+}_3 \rightarrow^{b^{-}_4} t^{-}_4)\&\,\varepsilon$$

for some $t^{-}_3 \leqslant t_3$, $b^{-}_3 \leqslant b_3$, $t^{+}_3 \geqslant t_3$, $b^{-}_4 \leqslant b_4$ and $t^{-}_4 \leqslant t_4$.

By assumption

$$(w_{11}, w_2) \rightsquigarrow (e_1, e_2) : (b_1, b_2)$$

and the induction hypothesis yields $t_3^{--} \leqslant t_3^-$, $t_{02}^- \leqslant t_{02}$, $b_3^{-\prime}$ and b_2' such that

$$cenv \vdash e_1 \mid t_3^{--} \,\&\, b_3^{-\prime} \quad \text{with } b_1; b_3^{-\prime} \leqslant b_3^-$$
$$cenv \vdash e_2 \mid t_{02}^- \,\&\, b_2' \quad \text{with } b_2; b_2' \leqslant b_{02}$$

It then follows that

$$cenv \vdash w_{12}\, e_1 \mid t_4^- \,\&\, (\varepsilon; b_3^{-\prime}; b_4^-) \quad \text{with} \quad b_1; \varepsilon; b_3^{-\prime}; b_4^- \leqslant b_3^-; b_4^-$$

and taking $t_{01}^- = t_4^-$ and $b_1' = \varepsilon; b_3^{-\prime}; b_4^-$ we have

$$cenv \vdash w_{12}\, e_1 \mid t_{01}^- \,\&\, b_1' \quad \text{with } b_1; b_1' \leqslant b_{01}$$
$$cenv \vdash e_2 \mid t_{02}^- \,\&\, b_2' \quad \text{with } b_2; b_2' \leqslant b_{02}$$

as well as $t_{01}^- \leqslant t_{01}$ and $t_{02}^- \leqslant t_{02}$.

The rule for rearranging the components. This case is straightforward (due to the symmetrical formulation of the proposition). □

B.5. Proof of Lemma 4.7

We proceed by induction on the structure of the evaluation context E.

The case $E ::= [\,]$. This is immediate since $t = t_0$ and $b = b_0$ and we may therefore take $t' = t_0'$ and $b' = b_0'$.

The case $E ::= E_0\, e$. Inspection of $cenv \vdash E[e_0] \mid t \,\&\, b$ reveals premisses of the form

$$cenv \vdash E_0[e_0] \mid t_1 \rightarrow^{b_3} t \,\&\, b_1, \qquad cenv \vdash e \mid t_1^- \,\&\, b_2$$

where $t_1^- \leqslant t_1$ and $b = b_1; b_2; b_3$. From the induction hypothesis we get

$$cenv \vdash E_0[e_0'] \mid t_1^+ \rightarrow^{b_3^-} t^- \,\&\, b_1'$$

where $t_1^+ \geqslant t_1$, $b_3^- \leqslant b_3$, $t^- \leqslant t$ and $b^\bullet; b_1' \leqslant b_1$. Taking $t' = t^-$ and $b' = b_1'; b_2; b_3$ we then have

$$cenv \vdash E_0[e_0']\, e \mid t^- \,\&\, b'$$

as well as the desired $t' \leqslant t$ and $b^\bullet; b' \leqslant b$.

The case $E ::= w\, E_0$. Inspection of $cenv \vdash E[e_0] \mid t \,\&\, b$ reveals premisses of the form

$$cenv \vdash w \mid t_1^+ \rightarrow^{b_2} t \,\&\, \varepsilon, \qquad cenv \vdash E_0[e_0] \mid t_1 \,\&\, b_1$$

where $t_1^+ \geqslant t_1$ and $b = \varepsilon; b_1; b_2$ and we also used Fact B.1. From the induction hypothesis we get

$$cenv \vdash E_0[e_0'] \mid t_1^- \,\&\, b_1'$$

where $t_1^- \leqslant t_1$ and $b^\bullet; b_1' \leqslant b_1$. Taking $t' = t$ and $b' = \varepsilon; b_1'; b_2$ we then have

$$cenv \vdash w\, E_0[e_0'] \mid t' \,\&\, b'$$

as well as the desired $t' \leqslant t$ and $b^\bullet; b' \leqslant b$.

The case $E ::= \texttt{let}\, i = E_0\, \texttt{in}\, e$. Inspection of $cenv \vdash E[e_0] \mid t \,\&\, b$ reveals premisses of the form

$$cenv \vdash E_0[e_0] \mid t_1 \,\&\, b_1, \qquad cenv[i \mapsto t_1] \vdash e \mid t \,\&\, b_2$$

where $b = b_1; b_2$. From the induction hypothesis we get

$$cenv \vdash E_0[e_0'] \mid t_1^- \,\&\, b_1'$$

with $t_1^- \leqslant t_1$ and $b^\bullet; b_1' \leqslant b_1$. From Lemma 4.3 we get

$$cenv[i \mapsto t_1^-] \vdash e \mid t^- \,\&\, b_2^-$$

where $t^- \leqslant t$ and $b_2^- \leqslant b_2$. Taking $t' = t^-$ and $b' = b_1'; b_2^-$ we then have

$$cenv \vdash \texttt{let}\ i = E_0[e_0']\ \texttt{in}\ e \mid t' \,\&\, b'$$

as well as the desired $t' \leqslant t$ and $b^\bullet; b' \leqslant b$. (Note that the "decrease" in behaviour may apparently take place "deeply embedded" in the behaviour rather than only at the front!)

The case $E ::= \texttt{if}\, E_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : t$. (Note that no confusion arises from using the type t here.) Inspection of $cenv \vdash E[e_0] \mid t \,\&\, b$ reveals premisses of the form

$$cenv \vdash E_0[e_0] \mid \texttt{bool} \,\&\, b_1, \qquad cenv \vdash e_1 \mid t_1 \,\&\, b_2, \qquad cenv \vdash e_2 \mid t_2 \,\&\, b_3$$

where $t_1 \leqslant t$, $t_2 \leqslant t$ and $b = b_1; (b_2 + b_3)$. From the induction hypothesis we get

$$cenv \vdash E_0[e_0'] \mid \texttt{bool} \,\&\, b_1'$$

where $b^\bullet; b_1' \leqslant b_1$ because the condition $t_4 \leqslant \texttt{bool}$ is equivalent to $t_4 = \texttt{bool}$. Taking $t' = t$ and $b' = b_1'; (b_2 + b_3)$ we then have

$$cenv \vdash \texttt{if}\, E_0[e_0']\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : t \mid t \,\&\, b'$$

as well as the desired $t' \leqslant t$ and $b^\bullet; b' \leqslant b$. □

B.6. Proof of Proposition 4.6

The proof is by cases on the rule used for the concurrent transition.

Sequential evaluation. In this case we have $\vec{b} = \varepsilon$ and $\vec{pi} = pi$ for some $pi \in \mathbf{PIdent}$ and

$$PP' = PP[pi \mapsto PP'(pi)], \quad cenv' = cenv, \quad PP(pi) \to PP'(pi)$$

From $cenv \vdash PP(pi) \mid PT(pi) \& PB(pi)$ and Proposition 4.1 we get $t' \leqslant PT(pi)$ and $b' \leqslant PB(pi)$ such that $cenv \vdash PP'(pi) \mid t' \& b'$. Taking

$$PT' = PT[pi \mapsto t'], \qquad PB' = PB[pi \mapsto b']$$

all conditions are satisfied.

Channel allocation. In this case we have $\vec{b} = b = t_0$ CHAN i_0 and $\vec{pi} = pi$ for some $pi \in$ **Pident** and

$$PP' = PP[pi \mapsto PP'(pi)], \qquad cenv' = cenv[ci \mapsto t_0 \texttt{ chan } i_0]$$

$$PP(pi) = E[\texttt{channel} : (\texttt{unit} \rightarrow^b t)\,()], \qquad PP'(pi) = E[ci]$$

where $ci \notin \mathrm{dom}(cenv)$ and $t = t_0 \texttt{ chan } i_0$. From

$$cenv \vdash E[\texttt{channel} : (\texttt{unit} \rightarrow^b t)()] \mid PT(pi) \& PB(pi)$$

we immediately get

$$cenv' \vdash E[\texttt{channel} : (\texttt{unit} \rightarrow^b t)()] \mid PT(pi) \& PB(pi).$$

Furthermore

$$cenv' \vdash \texttt{channel} : (\texttt{unit} \rightarrow^b t)\,() \mid t \& \varepsilon; \varepsilon; b, \qquad cenv' \vdash ci \mid t \& \varepsilon$$

so that using Lemma 4.7 we get t' and b' such that

$$cenv' \vdash E[ci] \mid t' \& b'$$

and where $t' \leqslant t$ and $b; b' \leqslant PB(pi)$. Taking

$$PT' = PT[pi \mapsto t'], \qquad PB' = PB[pi \mapsto b']$$

all conditions are satisfied.

Process creation. In this case we have $\vec{b} = b = t_0$ FORK b_0 and $\vec{pi} = pi_1, pi_2$ for some $pi_1, pi_2 \in$ **PIdent** and

$$PP' = PP[pi_1 \mapsto PP'(pi_1)][pi_2 \mapsto PP'(pi_2)]$$

$$cenv' = cenv,$$

$$PP(pi_1) = E[\texttt{fork} : (t \rightarrow^b \texttt{unit})\, w]$$

$$PP'(pi_1) = E[()], \quad PP'(pi_2) = w()$$

where $pi_2 \notin \mathrm{dom}(PP)$ and $t = \texttt{unit} \rightarrow^{b_0} t_0$. From

$$cenv \vdash E[\texttt{fork} : (t \rightarrow^b \texttt{unit})\, w] \mid PT(pi_1) \& PB(pi_1)$$

we get

$$cenv \vdash \texttt{fork} : (t \rightarrow^b \texttt{unit})\, w \mid \texttt{unit} \& \varepsilon; \varepsilon; b$$

and hence

$$cenv \vdash w \mid \texttt{unit} \rightarrow^{b_0^-} t_0^- \,\&\, \varepsilon$$

where $b_0^- \leqslant b_0$ and $t_0^- \leqslant t_0$ and we have used Fact B.1. Since

$$cenv \vdash () \mid \texttt{unit} \,\&\, \varepsilon$$

we get t' and b' from Lemma 4.7 such that

$$cenv \vdash E[()] \mid t' \,\&\, b'$$

and where $t' \leqslant PT(pi_1)$ and $b; b' \leqslant PB(pi_1)$. Taking

$$PT' = PT[pi_1 \mapsto t'][pi_2 \mapsto t_0^-], \qquad PB' = PB[pi_1 \mapsto b'][pi_2 \mapsto \varepsilon; \varepsilon; b_0^-]$$

all conditions are satisfied.

Synchronization. In this case we have $\vec{b} = b_1, b_2$ and $\vec{pi} = pi_1, pi_2$ for some $pi \in$ **PIdent** and

$$PP' = PP[pi_1 \mapsto PP'(pi_1)][pi_2 \mapsto PP'(pi_2)]$$
$$cenv' = cenv$$
$$PP(pi_1) = E_1[\texttt{sync} : (t_1 \,\texttt{com}\, b_1 \rightarrow^{b_1} t_1)\, w_1]$$
$$PP(pi_2) = E_2[\texttt{sync} : (t_2 \,\texttt{com}\, b_2 \rightarrow^{b_2} t_2)\, w_2]$$
$$PP'(pi_1) = E_1[e_1]$$
$$PP'(pi_2) = E_2[e_2]$$
$$(w_1, w_2) \rightsquigarrow (e_1, e_2) : (b_1, b_2)$$

For $j \in \{1,2\}$ we have

$$cenv \vdash E_j[\texttt{sync} : (t_j \,\texttt{com}\, b_j \rightarrow^{b_j} t_j)\, w_j] \mid PT(pi_j) \,\&\, PB(pi_j)$$

and get

$$cenv \vdash \texttt{sync} : (t_j \,\texttt{com}\, b_j \rightarrow^{b_j} t_j)\, w_j \mid t_j \,\&\, \varepsilon; \varepsilon; b_j$$

and hence

$$cenv \vdash w_j \mid t_j^- \,\texttt{com}\, b_j^- \,\&\, \varepsilon$$

where $t_j^- \leqslant t_j$ and $b_j^- \leqslant b_j$ and we have used Fact B.1.

Proposition 4.5 then gives t_1'', t_2'', b_1'' and b_2'' such that for $j \in \{1,2\}$ we have

$$cenv \vdash e_j \mid t_j'' \,\&\, b_j''$$

with $t_j'' \leqslant t_j^-$ and $b_j; b_j'' \leqslant b_j^-$. Using Lemma 4.7 we then get t_1', t_2', b_1' and b_2' such that for $j \in \{1,2\}$ we have

$$cenv \vdash E_j[e_j] \mid t_j' \,\&\, b_j'$$

with $t'_j \leqslant PT(pi_j)$ and $b_j; b'_j \leqslant PB(pi_j)$. Taking

$$PT' = PT[pi_1 \mapsto t'_1][pi_2 \mapsto t'_2], \qquad PB' = PB[pi_1 \mapsto b'_1][pi_2 \mapsto b'_2]$$

all conditions are satisfied (and b_1 and b_2 are as stated in Proposition 4.5). □

Acknowledgements

This work is partly funded by the DART-project (supported by The Danish Research Councils) and the LOMAPS-project (ESPRIT BRA 8130).

References

[1] D. Berry, R. Milner and D.N. Turner, A semantics for ML concurrency primitives, *Proc. POPL'92* (ACM, New York, 1992) 119–129.

[2] B. Berthomieu and T. LeSergent, Programming with behaviours in an ML framework: the syntax and semantics of LCS. *Proc. ESOP'94*, Springer Lecture Notes in Computer Science, Vol. 788, (Springer Berlin, 1994) 89–104.

[3] D. Bolignano M. Debabi, A coherent type system for a concurrent, functional and imperative programming language, *Proc. AMAST'93*, 1993.

[4] L. Cardelli and G. Longo, A semantic basis for quest. *J. Funct. Programming* **1** (4) (1991) 417–458

[5] G. Castagna, G. Ghelli and G. Longo, A calculus for overloaded functions with subtyping, *Proc. Lisp and Functional Programming 1992*, (ACM, New York, 1992) 182–192.

[6] M. Coppo and M. Dezani, A new type assignment for λ-terms, *Archiv. Math. Logik* **19** (1978) 139–156

[7] C. Crasemann, $\pi\lambda$-Kalküle für Prozesse und Funktionen, Ph.D. Thesis, Christian-Albrechts-Universität zu Kiel, 1992.

[8] B.A. Davey and H.A. Priestly, *Introduction to Lattices and Order* (Cambridge University Press, Cambridge, 1990).

[9] M. Felleisen and D.P. Friedman, Control operators, the SECD-machine, and the λ-calculus. in: ed., M.Wirsing, *Formal Descriptions of Programming Concepts III* (North-Holland, Amsterdam 1986) 193–219.

[10] A. Giacalone, P. Mishra and S. Prasad, Operational and algebraic semantics for Facile: a symmetric integration of concurrent and functional programming, *Proc. ICALP '90*, Springer Lecture Notes in Computer Science, Vol. 443 (Springer, Berlin, 1990) 765–780.

[11] K. Havelund and K.G. Larsen, The fork calculus, *Proc. ICALP '93*, Springer Lecture Notes in Computer Science, Vol. 700 (Springer, Berlin, 1993) 544–557.

[12] J.J.-Levy, B. Thomsen, L. Leth and A. Giacalone, Esprit basic research action 6454 – CONFER: CONcurrency and Functions: Evaluation and Reduction, *EATCS Bulletin*, Vol. 48 (1992) 88–106.

[13] J.M. Lucassen and D.K. Gifford, Polymorphic effect systems, *Proc. POPL'88*, (ACM, New York, 1988) 47–57.

[14] R. Milner, *Communication and Concurrency* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

[15] R. Milner, The polyadic π-calculus: a tutorial, Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991. Also in: eds. F.L. Bauer, W. Brauer, H. Schwictenberg, *Logic and Algebra of Specification*, (Springer, Berlin, 1993).

[16] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML* (MIT Press, Cambridge, CA, 1990).

[17] F. Nielson, The typed lambda-calculus with first-class processes, *Proc. PARLE'89*, Springer Lecture Notes in Computer Science, Vol. 366 (Springer, Berlin, 1989) 357–373.

[18] F. Nielson and H.R. Nielson, *Two-Level Functional Languages* (Cambridge University Press, Cambridge, 1992).

[19] F. Nielson and H.R. Nielson, From CML to process algebras (extended abstract), *Proc. CONCUR'93*, Springer Lecture Notes in Computer Science, Vol. 715 (Springer, Berlin, 1993) 493–508.

[20] H.R. Nielson and F. Nielson, Higher-order concurrent programs with finite communication topology, *Proc. POPL'94* (ACM, New York, 1994) 84–97.
[21] F. Nielson and H.R. Nielson, Constraints for polymorphic behaviours of concurrent ML, *Proc. CCL'94*, Springer Lecture Notes in Computer Science, Vol. 845 (Springer, Berlin,) 73–88.
[22] E.G.J.M.H. Nocker, J.E.W. Smetsers, M.C.J.D. van Eekelen and M.J. Plasmeijer, Concurrent clean, *Proc. PARLE'91*, Springer Lecture Notes in Computer Science, Vol. 506 (Springer, Berlin, 1991) 202–219.
[23] C. Reade, *Elements of Functional Programming* (Addison-Wesley, Reading, MA, 1989).
[24] J.H. Reppy, CML: a higher-order concurrent language, *Proc. PLDI'91* (ACM, New York, 1991) 293–305.
[25] J.H. Reppy, Higher-order concurrency, Ph.D. Thesis, Report 92-1285, Department of Computer Science, Cornell University, 1992.
[26] J.H. Reppy, Concurrent ML: design, application and semantics, *Proc. Functional Programming, Concurrency, Simulation and Automated Reasoning*, Springer Lecture Notes in Computer Science, Vol. 693 (Springer, Berlin, 1993) 165–198.
[27] D. Sangiorgi, A theory of bisimulation for the π-calculus, *Report ECS-LFCS-93-270*, Laboratory for Foundations of Computer Science, University of Edinburgh, 1993.
[28] J.-P. Talpin and P. Jouvelot, The type and effect discipline, *Proc. LICS'92* (Springer, Berlin, 1992) 162–173.
[29] B. Thomsen, A calculus of higher order communicating systems, *Proc. POPL'89*, (ACM, New York, 1989) 143–154.
[30] B. Thomsen, Plain CHOCS, *Report DOC 89/4*, Imperial College, University of London, 1989.
[31] B. Thomsen, Calculi for higher order communicating systems, Ph.D. Thesis, Imperial College, University of London, 1990.
[32] B. Thomsen, Polymorphic sorts and types for concurrent functional programs, Technical report ECRC-93-10, 1993.
[33] A. Wright and M. Felleisen, A syntactic approach to type soundness, Report TR 91-160, Dept. of Computer Science, Rice University, 1991.